Geomatics for the Built Environment
MSc Thesis

Vincent J. A. Vanderheeren

June 2026

Geomatics for the Built Environment
MSc Thesis

Pillar of Morphology

Enhancing point-based mathematical morphology for processing applications in heritage point clouds

Vincent Jean Arsène Vanderheeren

June 2026

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

Vincent Jean Arsène Vanderheeren: *Pillar of Morphology* (2026)

🅭🅯 This work is licensed under a Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.

The work in this thesis was carried out in the:

Geo-Database Management Centre
Delft University of Technology

Supervisors: Dr.ir Martijn Meijers
Dr.ir Ken Arroyo Ohori

Co-reader: Dr.ir Jesús Balado-Frías

Abstract

Heritage point clouds are an increasingly common tool used in the conservation and documentation of cultural assets. Yet, their geometric complexity, uniqueness, and scale make automated processing using machine learning or deep learning difficult. Mathematical morphology offers an alternative using geometric assumptions in the form of a structuring element, requiring no training data and remaining interpretable. This thesis extends the point-based mathematical morphology algorithms proposed by Balado et al. (2020) to make them useful for real-world heritage point cloud processing. The original algorithms are optimized using a parallel implementation on the GPU in the form of compute shaders, reducing the time complexity of both erosion and dilation to O(n) and achieving a speedup of approximately 100× over the sequential baseline. Four extensions to the algorithms are created: a continuous erosion score replacing the binary output, orientation-aware operations using TNB matrix rotation with both a predetermined and a brute-force variant, density-aware structuring elements in discrete and continuous forms, and a set of helper functions enabling compound morphological operations. These contributions are evaluated on three heritage applications. Segmentation using sequential morphological opening and hit-or-miss transform achieves a mIoU of 0.623 and an overall accuracy of 0.813 on the Images&PointClouds Cultural Heritage Dataset without parameter optimization. Object detection achieves F1 scores above 0.920 on the Fontana dei Mesi dataset. Gap filling via morphological closing performs well locally but is sensitive to the geometric complexity of the scene when applied globally, with edge detection and template matching providing more consistent results. The results show that point-based mathematical morphology, when extended with the contributions of this thesis, is a practical and effective tool for heritage point cloud processing that does not require annotated training data.

Keywords

POINT CLOUD PROCESSING · MATHEMATICAL MORPHOLOGY · CULTURAL HERITAGE · GPU · PARALLELIZATION · COMPUTE SHADERS · OPENGL · SEGMENTATION · OBJECT DETECTION · GAP FILLING · EROSION · DILATION · LIDAR · PHOTOGRAMMETRY

Acknowledgements

I would first like to thank my supervisors, Dr.ir. Martijn Meijers and Dr.ir. Ken Arroyo Ohori, for their guidance and advice throughout this process. A special thanks goes to Dr.ir. Jesús Balado-Frías for providing the initial framework for this thesis topic and for serving as co-reader during my defense.

My parents' support and genuine curiosity about my work, even when it became too complicated to explain, was still a very welcome source of encouragement. I am also very grateful to my friends for keeping me going during these last months, especially Alin, with whom I shared many long days of thesis writing, and my fellow students in the Geolab, whose shared struggle made the final stretches feel more bearable.

Finally, to my colleagues at CoffeeCompany: thank you for the coffee and matcha, and for listening to me complain about GPU shaders behind the bar, even though it was hard to relate to.

Contents

1 Introduction **2**
1.1 Motivation 2
1.2 Research objectives 3
1.3 Reading guide 4

2 Background **6**
2.1 Point clouds and the cultural heritage domain 6
2.2 Mathematical morphology in point clouds 7
2.3 Heritage applications of morphological operations 11
2.4 Base algorithm review 12

3 Method **18**
3.1 Optimization of the base algorithm 19
3.2 Density-aware operations 23
3.3 Orientation-aware operations 26
3.4 Helper and compound operations 29
3.5 Heritage applications 35
3.6 Datasets 39

4 Results **42**
4.1 Algorithm review 42
4.2 Heritage use case 59

5 Conclusion **70**
5.1 Main Findings 70
5.2 Relevance 71

6 Discussion **74**
6.1 Limitations 74
6.2 Future Work 76

Bibliography **78**

A Technical Implementation **82**
A.1 Code development 82
A.2 Data processing 87
A.3 Heritage use case set-up 88

B Algorithms **90**
B.1 Erosion algorithms 90
B.2 Dilation algorithms 94
B.3 Helper algorithms 96

C Declaration of AI usage **98**

List of figures

Figure 1.1 Dilation and erosion applied to 2D shapes. Input shape in red, structuring element in blue, result of the operation indicated by the blue dashed line. 2
Figure 2.1 Graphical representation of the compound morphological operations on 2D shapes, original shape in red, result of the first operation in blue, and the final resulting shape in yellow. 8
Figure 2.2 Voxel-based dilation using the Minkowski sum for dilation (Shah et al., 2022) . 9
Figure 2.3 Surface-based dilation using implicit functions (Calderon & Boubekeur, 2014) 10
Figure 2.4 Point-based erosion, source: (Balado et al., 2020) 13
Figure 2.5 Point-based dilation, source: (Balado et al., 2020) 13
Figure 2.6 Erosion with a 5×5 plane SE 14
Figure 2.7 Dilation 5-point line SE 14
Figure 2.8 Dilation of a plane with increasing density with a 10×10 plane set at the midway density 15
Figure 2.9 Opening (red) using a window structuring element on the HH dataset 15
Figure 3.1 Schematic overview of the developed algorithms, outlined algorithms are optional in compound operations. 18
Figure 3.2 Ranged nearest neighbor search in a 2D spatial hash grid, red depicts an average search, purple the most extreme case 20
Figure 3.3 Graphical representation of race conditions when inserting points in a spatial hash during dilation 22
Figure 3.4 Red points of which the structuring element extents do not overlap belong to one group 23
Figure 3.5 Self-adaptive structuring element generated from a circle by discretely subsampling at different minimum distances 23
Figure 3.6 Self-adaptive structuring element generated from a circle by continuously assigning distances to points. 24
Figure 3.7 Graphical representation of the erosion score algorithm, the input (red) gets assigned a score by dividing the matches by the total number of points in the SE (blue) 25
Figure 3.8 Direction of the rotation for an arrow on the xz-plane pointing in the direction of the z-axis 27
Figure 3.9 Spherical coordinate system with local basis vectors, source: (Ag2gaeh, 2008) 28
Figure 3.10 Graphical representation of boolean operations applied to point clouds 29
Figure 3.11 Approximation of the surface-based dilation obtained by subtracting two dilations using spherical SEs one threshold distance apart in radius. 30

Figure 3.12 Points arranged on a sphere using the Fibonacci spiral, image source: (cchu79, 2023) ... 32
Figure 3.13 Graphical representation of the applications of mathematical morphology ... 35
Figure 3.14 Structuring element used for edge detection, points used in the positive erosion (red) complement those used in the negative (blue) ... 38
Figure 3.15 Stanford Armadillo and Bunny point clouds ... 39
Figure 3.16 Datasets from OpenHeritage3D, source: (CyArk, 2019; Kress Foundation, 2021; Teppati Lose et al., 2023) ... 40
Figure 3.17 Datasets from Images&PointClouds Cultural Heritage, source: (Pellis et al., 2025a) ... 40
Figure 3.18 Images of the Images&PointClouds sites next to their colored labels, source: (Pellis et al., 2025a) ... 41
Figure 4.1 Total runtime and time per SE point for the base erosion algorithm compared to the new parallel implementation ... 43
Figure 4.2 Total runtime and time per SE point for the erosion variants, total time on the left, time per point on the right ... 44
Figure 4.3 Total runtime and time per SE point for the score erosion with structuring elements of varying sizes and shapes ... 45
Figure 4.4 Total runtime and time per SE point for the brute-force orientation erosion for different numbers of sampled rotations. Pale dot-dash lines indicate the approximate time a regular erosion would require for the same rotation count ... 45
Figure 4.5 Total runtime and time per SE point for the density-aware dilation as a function of combined SE size, for an increasing number of density increments ... 46
Figure 4.6 Total runtime and time per SE point comparison between the sequential and parallel dilation on a hollow cube with a 5-point line SE ... 47
Figure 4.7 Total runtime and time per SE point of the dilation variants for increasing input size, evaluated on a hollow cube with a 10×10 plane SE ... 48
Figure 4.8 Total runtime and time per SE point for the dilation for a line and plane SE of increasing size, evaluated on a 1M point hollow cube ... 49
Figure 4.9 Total runtime and time per SE point for the density-aware dilation as a function of combined SE size, for an increasing number of density increments ... 50
Figure 4.10 Mathematical validity test: hollow cube (101,402 points) eroded with a 5×5 plane SE, resulting in the expected 32,258 points ... 51
Figure 4.11 Erosion of the Stanford Armadillo with a 5-point line SE. The original sequential implementation (left) versus GPU-accelerated implementation (right), both resulting in 16,247 points ... 51
Figure 4.12 Binary vs. score erosion of the Stanford Bunny with a 5-point line SE. Score is mapped from blue (low) through red (perfect match) ... 52
Figure 4.13 Density erosion on a density plane: discrete (left) and continuous (right) for both binary and score variants. Red indicates high match score, yellow medium, blue low ... 52

Figure 4.14 Orientation-aware erosion on a hollow cube: full orientation erosion (left), brute-force restricted to z-axis rotations (centre), brute-force with z-axis and tilt (right) . . . 53
Figure 4.15 Close-up of grouping output; each color represents a distinct group . . . 53
Figure 4.16 Mathematical validity test: hollow cube (101,402 points) dilated with a 5×5 plane SE, resulting in the expected 371,850 points . . . 54
Figure 4.17 Nearest-neighbour distance histogram for the dilation output. The red line marks the 0.25 threshold; 1.18% of points fall below it . . . 55
Figure 4.18 Difference between the original (red) and the parallel (blue) dilation outputs, with shared points colored purple . . . 55
Figure 4.19 Density-aware dilation on a density plane: discrete (left) and continuous (right) . . . 56
Figure 4.20 Orientation-aware dilation and closing on a hollow cube with holes: input with estimated normals (left), dilation result (center), closing result (right) . . . 56
Figure 4.21 Score erosion of the Stanford Bunny with a 5-point line SE at increasing distance thresholds. Blue indicates a low score, red a high score . . . 57
Figure 4.22 Segmentation results for 5_CB: ground truth (left) and predicted (right) . . . 60
Figure 4.23 Pillar segmentation obtained with the opening for 5_CB: ground truth (left) and opening result (right) . . . 61
Figure 4.24 Pillar detection in the FM dataset using the hit-or-miss transform. Red indicates detected pillars . . . 62
Figure 4.25 Shell detection in the FM dataset using the brute-force orientation opening. Red indicates detections. . . . 63
Figure 4.26 Window detection in the HH dataset using the orientation-aware hit-or-miss transform . . . 63
Figure 4.27 Acorn detection on the HH facade using the density-aware opening: discrete (top) and continuous (bottom) . . . 64
Figure 4.28 Gap filling visual results for the PB dataset using a 15-point line SE in the x-direction. Red indicates inserted points . . . 67
Figure 4.29 Edge detection results on the PB dataset. Left: detected edges in the positive x-direction overlaid on the point cloud. Right: comparison of positive x-direction detections (blue) against ground truth edges of all directions (red) . . . 68
Figure 4.30 Template matching applied to the PB dataset. Left: input with missing column data. Right: reconstructed points added by template matching shown in red. . 68

List of tables

Table 1 Information for the OpenHeritage3D Datasets 40
Table 2 Statistics for the Images&Points Cultural Heritage Dataset 41
Table 3 Segmentation metrics for the Images&PointClouds dataset 59
Table 4 Object detection metrics for the FM and HH datasets with different objects 65
Table 5 Results of closing with different SEs for gap-filling in the PB point cloud 66
Table 6 Results of orientation-aware erosion with different SEs for edge detection in the PB point cloud 67
Table 7 Hardware used for benchmarking and the heritage use case 82

List of algorithms

Algorithm 1 erode-old 90
Algorithm 2 erode 91
Algorithm 3 erode-score 91
Algorithm 4 erode-density-score 92
Algorithm 5 erode-orientation-score 93
Algorithm 6 erode-orientation-score-brute 93
Algorithm 7 dilate-old 94
Algorithm 8 dilate 94
Algorithm 9 dilate-density 95
Algorithm 10 dilate-orientation 95
Algorithm 11 add 96
Algorithm 12 subtract 96
Algorithm 13 intersect 96

1

Introduction

1.1 Motivation

The wealth of cultural heritage sites found across the world is a testament to human creativity and ingenuity throughout history. Yet, it is precisely this uniqueness of every site that makes automated processing so difficult. As LiDAR scanning and photogrammetry make capturing 3D heritage point clouds increasingly common in the conservation of our cultural assets (Gil et al., 2024), the need for tools to efficiently process this rich source of geometric and radiometric data grows (Nebiker et al., 2010).

A defining obstacle in processing heritage point clouds is that they feature geometrically complex, handcrafted, and non-serialized objects (Matrone et al., 2020; Pierdicca et al., 2020). This obstructs the application of data-driven methods such as machine learning and deep learning, which have otherwise revolutionized urban and indoor point cloud processing (Gil et al., 2024). The lack of large annotated training data and the difficulty in finding statistical patterns in this heterogeneous data (Grilli et al., 2019; Matrone et al., 2020) necessitate the search for alternative processing techniques that do not rely on these preconditions.

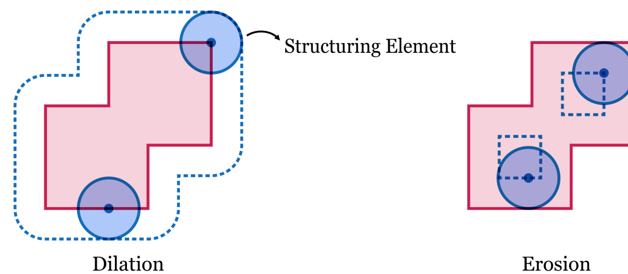


Figure 1.1 *– Dilation and erosion applied to 2D shapes. Input shape in red, structuring element in blue, result of the operation indicated by the blue dashed line.*

As opposed to relying on statistical learning, which does not generalize well to heritage data, geometry-based processing techniques provide an attractive alternative. One such technique is mathematical morphology, developed at the École des Mines de Paris by Serra (1982). This technique is able to perform tasks such as segmentation, object detection, and gap filling by applying its two main operators: dilation and erosion, shown in Figure 1.1. Since this technique

uses a structuring element to encode the geometric assumptions about target objects, it can easily be tailored to unique datasets.

The research from Balado et al. (2020) proposes a method for applying mathematical morphology directly to 3D point clouds without any intermediate surface reconstructions or rasterization. However, their method was only validated on small, synthetic, or simple real-world datasets. Heritage point clouds, on the other hand, often consist of tens of millions of points (Matrone et al., 2020), acquired at varying densities and with objects appearing in multiple orientations. Noise introduced during the capturing process adds an additional challenge, for which the binary output of the erosion is unequipped to deal with. Since these algorithms run sequentially on the CPU, they possess insufficient processing speed required at this scale. All these issues combined make it clear that the current state of this method lacks the properties necessary to effectively process large, unique heritage point clouds.

This thesis first explores the shortcomings of the method proposed by Balado et al. (2020), after which it will address the identified improvement areas by implementing algorithm extensions, utilizing GPU parallel processing with compute shaders. Finally, these improved algorithms are tested on real-world heritage datasets for segmentation, object detection, and gap filling use cases.

1.2 Research objectives

The objective of this thesis is summarized in the following research question:

How can point-based mathematical morphology be enhanced and effectively applied to enable segmentation, object detection, and gap filling in heritage point clouds?

This research question is further developed in two subquestions, the first one covering the technical implementation, while the second one focuses on the heritage applications:

∴ *How can the dilation and erosion algorithm by Balado et al. (2020) be extended and accelerated to support segmentation, object detection, and gap filling on heritage point clouds?*

∴ *What performance and accuracy do the improved algorithms achieve during segmentation, object detection, and gap filling in heritage point clouds?*

The first subquestion establishes the starting point of the research. The algorithm proposed by Balado et al. (2020) serves as the foundation for new improvements to be added to. Its limitations regarding processing heritage point cloud are identified through implementation and analysis, covered in Section 2.4. For each identified limitation, different approaches are explored, and a solution is implemented. The improvements are then benchmarked, validated, and tested for robustness across a variety of inputs to verify that they produce the expected results. The discussion of each improvement also notes how the changes alter the output of the algorithms. The implementation is discussed in Section 3.1 - Section 3.4, and the results are described in Section 4.1.

The second subquestion returns to the heritage use cases with the improved algorithms and evaluates the result using different metrics and visual analysis. These results are then compared to the baseline. Performance and accuracy are measured for segmentation, object detection, and gap filling. The set-up of the use cases is explained in Section 3.5 - 3.6, and the results are found in Section 4.2.

1.3 Reading guide

Chapter 2 introduces the background necessary to understand the contributions of this thesis. It starts with an overview of previous work on point cloud processing and how it is used in the heritage domain, followed by the theory of mathematical morphology and its existing adaptations to point cloud data. The chapter ends with a review of the base algorithm by Balado et al. (2020). It establishes a performance baseline and identifies the limitations that motivate the improvements developed in this thesis.

Chapter 3 describes how the base algorithm is extended and how it is applied to the three heritage use cases of segmentation, object detection, and gap filling. It explains the changes to the algorithm, discusses the design choices behind each application, and introduces the datasets used for evaluation.

Chapter 4 presents the results of the algorithm review and the heritage use cases. The algorithm review evaluates the performance and accuracy of the enhanced implementation compared to the baseline. The heritage use case results then show the effectiveness of the improvements on real heritage point clouds.

Chapter 5 summarizes the main findings of the thesis and answers the research questions introduced in this chapter. It also highlights the relevance of this research in a broader context.

Chapter 6 reflects on the implications of the results, discusses the limitations of the proposed approach, and indicates possible directions for future work.

Finally, three appendices are included. Appendix A documents the technical implementation. It covers the code development, data processing, and the setup of the heritage use case. Appendix B provides the pseudocode of each algorithm developed in this thesis. Appendix C contains the declaration of AI usage.

1.3 Reading guide

2 Background

First, the role of point clouds in the cultural heritage domain will be highlighted alongside some more general properties of point clouds. Afterwards, mathematical morphology itself and the ways it has been previously applied to point clouds will be discussed. Then, the operations of mathematical morphology will be related to various use cases in the heritage domain and how it differs from other methods. Finally, the algorithms proposed for point-based erosion and dilation by Balado et al. (2020) will be reviewed.

2.1 Point clouds and the cultural heritage domain

Point clouds are increasingly used for heritage documentation and analysis, but before discussing their specific applications, it is worth examining what makes them a preferred data format in the first place.

2.1.1 The rich point cloud paradigm

The *rich point cloud paradigm* is a term coined by Nebiker et al. (2010) and refers to a new way to approach 3D city modeling. It differentiates itself from geometric 3D modeling and image-based modeling in four distinct ways:

- It is a **better**, more accurate, and detailed capture of the 3D urban environment, both geometrically as well as radiometrically.
- It is a **faster** way of creating and updating 3D city models via laser scanning.
- It is a **cheaper** model for large urban environments, especially when only a graphical visualization is required.
- It is a **smarter** way of automatically creating semantically rich 3D city models.

Considering these benefits, methods for processing directly on point clouds will become more relevant in the future for applications within the urban environment, and the cultural heritage domain is no exception.

2.1.2 Point cloud challenges

Despite these advantages, point clouds come with inherent challenges. Other than the higher processing power requirements, the lack of spatial context and empty space between the 3D points, as well as the spatially biased data density and sampling of 3D point cloud data, provide further difficulties in point cloud processing (Nebiker et al., 2010).

Many techniques exist to mitigate these issues, but rather than seeing these aspects as problems, they should be viewed as fundamental properties of point cloud datasets, and methods working directly on point clouds should be able to robustly handle them.

2.1.3 Point cloud capture

Point clouds are acquired through two main techniques. Photogrammetry reconstructs geometry from overlapping images, either captured terrestrially or from drones, and it naturally collects color information. LiDAR-based acquisition has a larger range of capturing methods. Terrestrial Laser Scanning provides the highest geometric accuracy and is often applied to scan individual buildings, although it is time-consuming and prone to occlusions. Airborne Laser Scanning provides a more rapid coverage of large heritage landscapes and can penetrate vegetation, but produces lower-density point clouds, which are essentially 2.5D. Mobile Laser Scanning, mounted on vehicles or robots, balances speed and accuracy for large-scale outdoor documentation. Finally, handheld scanners are used for small artefacts or narrow spaces where other methods do not work (Yang et al., 2023).

However, no technique covers all requirements at the same time, and multi-platform data fusion is commonly used to obtain point clouds that are both spatially complete and multi-resolution (Yang et al., 2023). The characteristics of the resulting data, including point density, like the availability of color, and the level of noise, can vary a lot between techniques, which directly affects the performance of any further processing (Cera et al., 2025).

2.1.4 Point clouds for heritage

3D point clouds have become a core data format in the documentation and analysis of cultural heritage. Their ability to capture complex geometry at high spatial resolution makes them very applicable to a domain that is characterized by its irregular forms, intricate surface detail, and a wide range of spatial scales (Yang et al., 2023). The main use cases include digital documentation and archiving, the creation of Heritage Building Information Models (HBIM), conservation and structural analysis, like damage investigation and finite element modelling, immersive visualization, and GIS-based landscape planning (Yang et al., 2023).

Many of these applications depend on the ability to extract semantic information from the raw point cloud. Automatic labelling of architectural elements for HBIM, detection of structural defects for conservation purposes, and cleaning or completion of point clouds for virtual experiences all require some form of segmentation, object detection, or gap filling.

2.2 Mathematical morphology in point clouds

As defined by Serra (1982), mathematical morphology is a set-theory processing technique for the analysis of geometrical structures, originally developed for binary images and later extended to grayscale images and more general spaces. The mathematical background of this theory will be further explained in this section.

Mathematical morphology can be applied to point clouds through several strategies. Since point clouds are often considered unstructured raw data, they are typically converted into alternative

representations before applying morphological operators. One common approach is rasterization, where the point cloud is mapped to a 2D image or a 3D voxel grid. Another option is surface reconstruction, using the resulting mesh as input for morphological processing. Finally, morphological operators can be defined directly on point sets or graphs derived from point clouds, which aligns with the rich point cloud paradigm and preserves the original geometric information. These three methods will be further explored later in this section.

2.2.1 Mathematical background

Let $X \subseteq \mathbb{Z}^n$ be a binary set representing the input object, and let $B \subseteq \mathbb{Z}^n$ be a structuring element. The structuring element gets translated along the elements of X, with a translation vector $b \in B$, let $X_b = \{x + b \mid x \in X\}$ denote the translation of X by b.

Erosion removes the elements of the input object that cannot fully contain the structuring element, thereby shrinking it. Erosion keeps only those points of X for which the translated structuring element B is entirely contained in X. It is formally defined as:

$$X \ominus B = \bigcap_{b \in B} X_b \tag{2.1}$$

Dilation describes the expansion of a set by adding points where the structuring element overlaps with X. Dilation enlarges X by adding points within the reach of the structuring element. It is formally defined as:

$$X \oplus B = \bigcup_{b \in B} X_b \tag{2.2}$$

Erosion and dilation are dual operations. For a symmetrical structuring element, the erosion of the complement of the input object is equal to the complement of its dilation. Let X^c denote the complement of X and B^s the symmetric reflection of B. Their duality is expressed as:

$$X^c \ominus B = (X \oplus B^s)^c \tag{2.3}$$

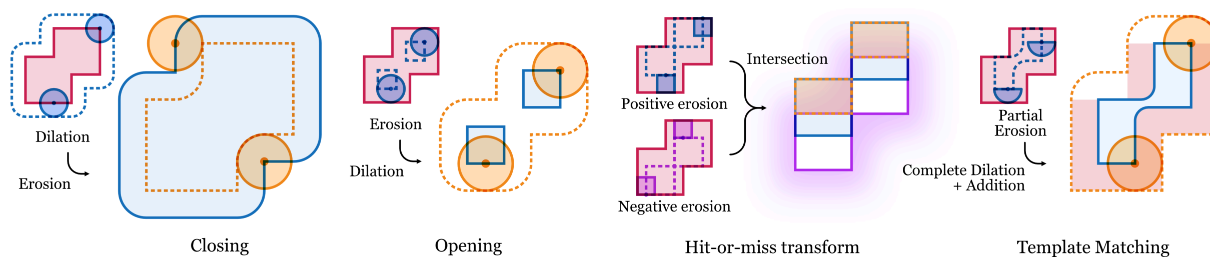


Figure 2.1 *– Graphical representation of the compound morphological operations on 2D shapes, original shape in red, result of the first operation in blue, and the final resulting shape in yellow.*

Opening is defined as erosion followed by dilation using the same structuring element. Opening removes small objects and narrow connections while preserving the overall shape and size of larger structures:

$$X \circ B = (X \ominus B) \oplus B \tag{2.4}$$

Closing is defined as dilation followed by erosion. Closing fills small holes and gaps, and smooths object boundaries:

$$X \bullet B = (X \oplus B) \ominus B \tag{2.5}$$

The hit-or-miss transform is a pattern detection operator used to identify specific configurations within a binary set. Let C and D be two structuring elements satisfying $C \cap D = \emptyset$. The transform is defined as:

$$X \odot B = (X \ominus C) \cap (X^c \ominus D) \tag{2.6}$$

It can be interpreted as the simultaneous erosion of the object and its complement, enabling precise detection of compound shapes.

Template matching is a custom operation used in this thesis; here it is denoted with the $\circledast$. Let B and B^p be two structuring elements satisfying $B^p \subsetneq B$. The template matching is defined as:

$$X \circledast B = X \cup ((X \ominus B^p) \oplus B) \tag{2.7}$$

where the erosion by B^p identifies points whose local neighborhood satisfies the partial structuring element, and the following dilation by the full structuring element B reconstructs the complete configuration around each such point. The result is the union of the original point set with all reconstructed configurations.

2.2.2 Raster-based

Raster-based methods first convert the point cloud into a 2D or 3D raster. The resolution of the raster is important, as a too coarse raster will result in data loss, while a too fine raster will necessitate the interpolation of missing cells.

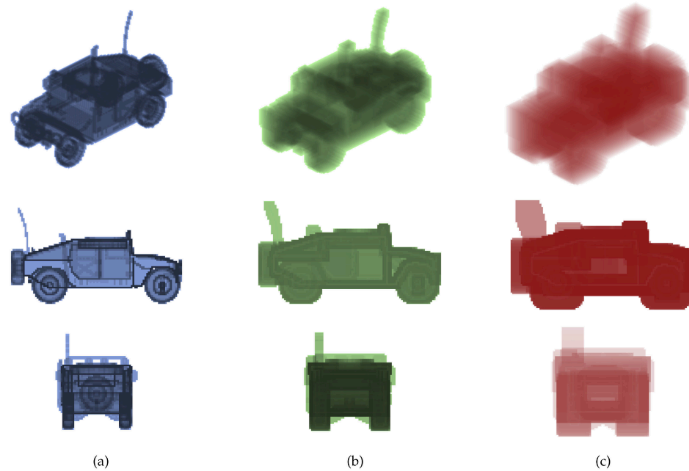


Figure 2.2 – *Voxel-based dilation using the Minkowski sum for dilation (Shah et al., 2022)*

Raster-based morphology is often used for DTM filtering. By converting a point cloud to a DTM raster, morphological operations can segment the dataset into ground and non-ground points. (Li et al., 2017). Another use case is presented by Shah et al. (2022). It takes a point cloud as

input, then voxelizes it and applies a Minkowski sum to it for collision detection. Since it is a lightweight method, it can be performed in real-time.

Raster-based operations are well-defined, fast, and easy to implement, but often simplify the data. They also suffer from grid alignment, in which straight lines might be interpolated in a jagged way, for different rotations.

Generally, these rasters are not converted back into point clouds, although this would be a simple operation, taking the center of each grid cell as one point in the point cloud.

2.2.3 Surface-based

Surface-based methods rely on having access to the surface mesh of the object to apply mathematical morphology. The method presented by Lien (2008) uses point clouds as an intermediate step to apply mathematical morphology to a mesh. The normal vector of the input mesh can therefore inform the direction of the morphology to apply its three filters to obtain an erosion or dilation.

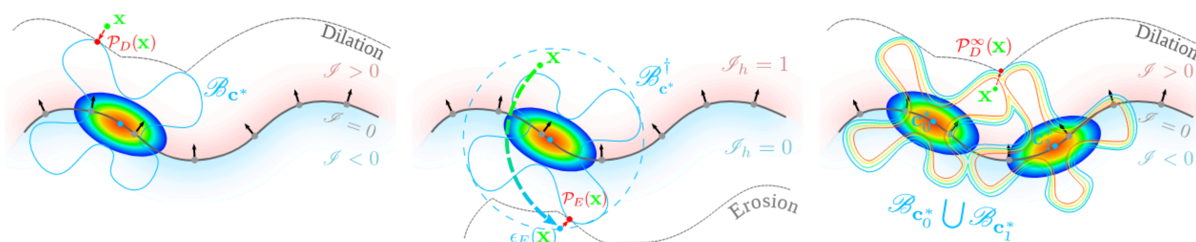


Figure 2.3 *– Surface-based dilation using implicit functions (Calderon & Boubekeur, 2014)*

Calderon & Boubekeur (2014) propose using implicit functions fitted to the point cloud to calculate the erosion and dilation. The obtained surface is then interpolated back into a point cloud.

While these methods are able to obtain good results, their reliance on having access to a surface mesh — or the implicit representation of it — causes the method to poorly handle messy real-world data, which is not always volumetrically enclosed. Surface-based mathematical morphology is subject to the same issues present when converting a point cloud into a 3D model, which the rich point cloud paradigm is trying to circumvent. Furthermore, it does not present solutions to applications like segmentation and gap filling, which are common applications of image-based mathematical morphology.

2.2.4 Point-based

Point-based mathematical morphology applies erosion and dilation directly on the points, using a threshold distance. The result is slightly different from the other two approaches, as the point-based approach does not take into consideration the 3D object, but only the points on the hollow outer shell. A dilation will therefore make the shell thicker, rather than move it further out (Balado et al., 2020). Because of the difference in output, this approach can be used for applications like segmentation and gap filling.

This research will take this method as a basis for further technical developments. As such, it is explained in more detail in Section 2.4.

2.3 Heritage applications of morphological operations

Mathematical morphology has many different applications. While they are generally applicable to all point clouds, they are particularly interesting for the heritage datasets. First, point cloud data of heritage sites is readily available, as LiDAR scanning is one of the preferred ways of digitally storing these sites. Second, heritage sites often feature unique elements not present anywhere else, making machine learning applications harder to train (Matrone et al., 2020).

Below, several applications are further explored in their relation to heritage.

2.3.1 Segmentation

Segmentation consists of labelling all points in a dataset with a predefined category. During this thesis, the Images&PointClouds dataset (Pellis et al., 2025a) is used to validate segmenting results within the heritage domain. It includes heritage-specific classes like columns and vaults, among more typical ones like roofs and walls.

Traditional methods to segment point clouds, like region growing, Hough transform, and RANSAC, rely on the mathematical constraints of the point cloud to perform their operations. They are generally computationally expensive and need manual tuning to obtain accurate results (Yang et al., 2023).

Recently, machine learning and deep learning have been applied to heritage point clouds. They are able to obtain superior results with more difficult geometries. However, machine learning requires extensive feature engineering. An example of this is found in the paper by Grilli et al. (2019), where geometric properties such as sphericity and surface variation are combined with manual labels to enable a random forest model to predict classes. It was able to obtain good segmentation results, although the datasets with the best results were structures that can be easily segmented along horizontal planes. As a result, it lost predictive power when excluding z-coordinates.

Deep learning, on the other hand, has high adaptability but requires more input data and larger training periods. Also, it is more of a “black box”, which produces results that are hard to fully interpret (Matrone et al., 2020). The paper that introduced the dataset (Pellis et al., 2025b) used in this thesis uses a deep learning approach to segment images of original structures and reprojects them afterwards to obtain a point cloud segmentation, obtaining good results when training on individual structures, but poor transfer between different ones.

The method proposed by Frías et al. (2020) using mathematical morphology, specifically the opening operation, is more similar to the traditional segmentation methods, as it needs to be manually tuned to accurately segment elements like walls and curbs, although a parallel implementation would reduce the long processing times generally associated with these methods.

2.3.2 Object detection

Similar to segmentation, object detection aims to find specific structures in point clouds. But rather than giving all points a label, it focuses on finding a specific object, resulting in a binary output rather than a segmented one.

Previtali et al. (2013) suggests using the periodical spacing of many objects, like windows, as a way to inform object detection. However, this method is restricted to only such periodically spaced objects. Another approach is machine learning on the geometric qualities of the point cloud (Moyano et al., 2021), but just like segmentation, it relies on the quality of the training data, which is hard to generalize for heritage data.

The paper from Balado et al. (2020) depicts a method based on mathematical morphology, through the opening operation. Nevertheless, it is limited due to the manual matching of the exact density and orientation of the object.

2.3.3 Gap filling

Gap filling generally consists of two tasks: hole detection and surface reconstruction.

Many different ways exist to detect holes in point clouds. Tabib et al. (2023) first performs a periphery detection, similar to erosion, to detect points that are located on an edge. Using a machine learning classifier, it separates edges from holes. Liu et al. (2022) makes use of 2D α-shape boundaries and relative position matching, while Ren et al. (2022) performs a triangular mesh analysis.

While surface reconstruction is often done using a deep learning approach (Feng et al., 2024; Li et al., 2024; Ren et al., 2022; Tabib et al., 2023), they are often limited in the point size of the scenes they want to reconstruct due to the increasing training time required. They often focus on individual objects (Ren et al., 2022; Tabib et al., 2023), or small subsections (Feng et al., 2024; Li et al., 2024). Liu et al. (2022) provides a method for mathematical morphology to be used as an alternative, although the method is restricted to planar surfaces.

2.4 Base algorithm review

2.4.1 Current algorithm

The method for applying point-based dilation and erosion — the building blocks of mathematical morphology — is based on the method described by Balado et al. (2020). The method takes a similar approach to the raster-based method covered in Section 2.2.2. A key difference is that mathematical points have zero area, so a collision cannot occur. To resolve this, a certain threshold area is used around every point. This means that if the algorithm checks for collision, it checks if there are any points present within the threshold distance by finding the nearest neighbor and checking if the distance is below the threshold.

The erosion algorithm loops over every point of the input data and overlays the structuring element at that point. It then checks for every point of the translated structuring element if a

point is present in the input point cloud. If this is true for every point of the structuring element, the point gets added to the eroded point cloud.

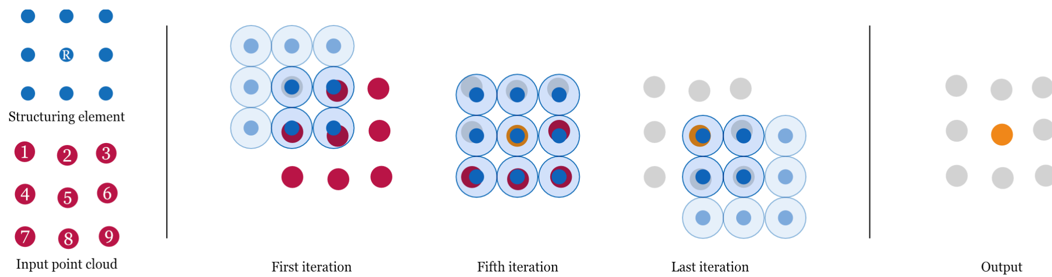


Figure 2.4 *– Point-based erosion, source: (Balado et al., 2020)*

For the dilation, the algorithm also loops over every point of the input data and overlays the structuring element at that point. Then, for every point of the translated structuring element, it checks if a point of the dilated point cloud is not yet present there. If there is no conflict, the point gets added to the dilated point cloud.

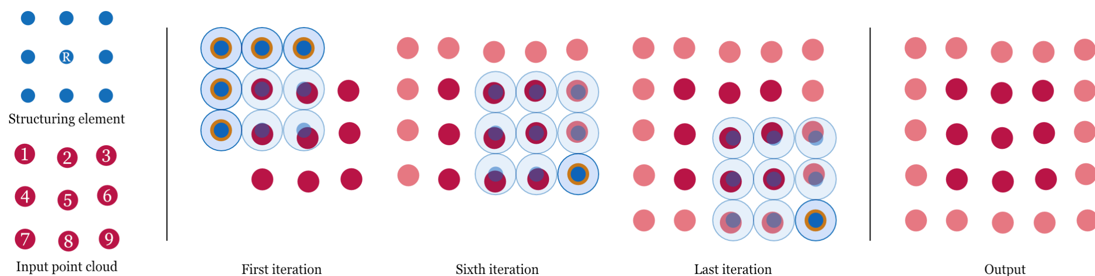


Figure 2.5 *– Point-based dilation, source: (Balado et al., 2020)*

This algorithm is currently implemented in MATLAB, but since the rest of this project will be coded in C++ using the CGAL library, the algorithm is first rewritten. This ensures that when the performance is compared to the improved model, the differences are actually due to the improvements made to the algorithm rather than the difference in programming language.

2.4.2 Performance evaluation

The current algorithm will be tested on efficiency, namely the time it takes to execute operations on different-sized point clouds with differently sized structuring elements, and how the time complexity scales.

Afterwards, the algorithm will be used to perform more complex tasks related to the heritage use case to identify where improvements are necessary to better perform these tasks.

This testing will allow the creation of a baseline performance to compare the enhanced algorithm against, as well as a basis for the proposed improvements.

Processing time

The current computational performance of the erosion and dilation algorithm is insufficient for processing large point clouds. Erosion exhibits a time complexity of O(n log n), as it uses kD-tree nearest-neighbor queries, where the operations like search, insert, and delete have an

average time complexity of O(log n) and worst-case O(n) (Bentley, 1975a). While this scaling could be manageable, practical testing shows that erosion already exceeds one second for 100K points, with a relatively small structuring element of 100 points, as shown in Figure 2.6. This means that point clouds of tens of millions of points, commonly obtained from laser scanners, would require minutes to process. Although this is still not the worst performance for erosion in isolation, any extension to the algorithm to include density-aware and orientation-aware processing will increase the processing time necessary per point, further deteriorating performance. This shows the need to further optimize the erosion algorithm to create a strong foundation for any additional extensions.

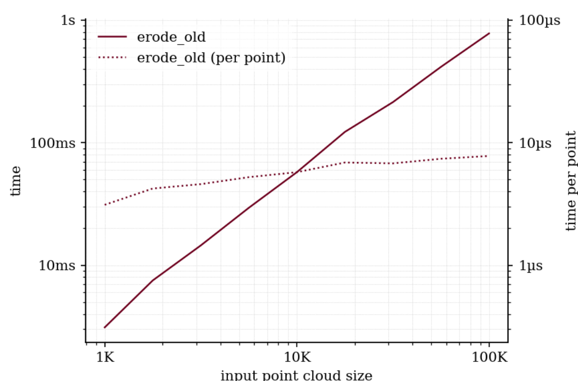


Figure 2.6 – *Erosion with a* 5×5 *plane SE*

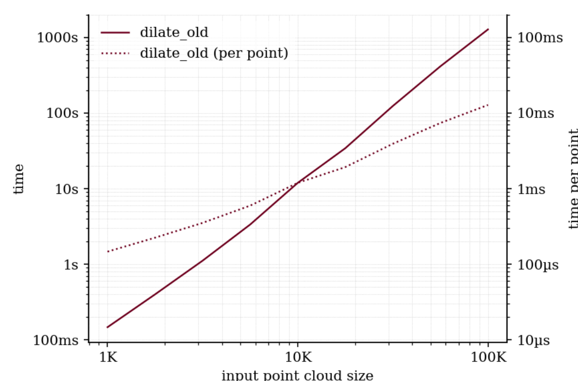


Figure 2.7 – *Dilation 5-point line SE*

Dilation, on the other hand, presents a more severe processing bottleneck, showing $O(n^2)$ time complexity. Each candidate point is tested against an ever-growing set of accepted points, due to which every insertion requires more processing time as the algorithm continues. This, combined with a poorly balanced tree resulting from naively inserting, causes the processing time to scale quadratically with the input size. This is confirmed by the observations presented in Figure 2.7, where dilating only 100K points with a structuring element of five points already requires more than 16 minutes. This result makes the current algorithm unusable for any real-world point clouds. To decrease the runtime, a serious overhaul of the dilation algorithm is required, in which the spatial structure used to store the accepted points has a better insertion time complexity and does not suffer from naive insertion.

Applicability to real-life datasets

The current algorithms for erosion and dilation on point clouds have several limitations when applied to real-world heritage data, particularly in the context of segmentation, object detection, and gap filling.

A first issue is related to varying densities present in point clouds. When performing dilation on a point cloud with varying density, such as the gradient density plane shown in Figure 2.8, the output does not match the local density of the input. On the high-density side, the edge between the original point clouds and the dilated points is harsh and clearly visible as the density drops abruptly. The low-density area, however, gains new points, making it artificially almost equally

dense as the structuring element. While this behavior might be desired for some applications, it means that the original spatial distribution is not preserved, which can be a drawback in point clouds where the density can say something about the way the data was acquired. Furthermore, in the context of gap filling, this will make the filled areas stand out within the point cloud, rather than seamlessly blending in.

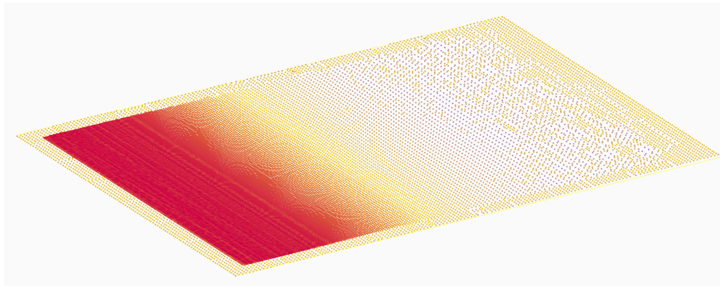


***Figure 2.8** – Dilation of a plane with increasing density with a 10×10 plane set at the midway density*

During object detection using erosion, the choice of a structuring element becomes difficult for a point cloud with varying density. A structuring element that is denser than the input will often fail to find matches, resulting in few or even no detections in low-density areas, even when the target geometry is present. Using a structuring element that is less dense than the input, on the other hand, lacks detail to reliably detect objects. This problem is exacerbated by the binary nature of the output. A point either survives or does not, meaning that points get discarded, even if they almost satisfy the structuring element. These near-misses, however, are still valuable as noisy real-world data introduces small local deviations. A near-miss, therefore, might not be caused by the wrong geometry, but rather by measuring errors.

***Figure 2.9** – Opening (red) using a window structuring element on the HH dataset*

The result of applying erosion to the Houghton Hall point cloud with a window pane structuring element shows these problems clearly in Figure 2.9. The noise in the scan causes several windows to go undetected, while many sections of the surrounding wall are falsely identified because the hollow interior of the window pane structuring element matches the wall surfaces as well as the window frames. Additionally, the erosion is axis-aligned, meaning it only detects

windows that share the same orientation as the structuring element. Windows on the left and right outer walls of the building are entirely missed, as shown by the absence of detections outside the front facade.

Finally, the current algorithm only evaluates whether a structuring element matches the input geometry, without taking into account empty space. Detecting objects that are also defined by their emptiness, such as window openings, doorways, or recessed niches, requires the ability to distinguish between a region that lacks points due to occlusion or noise and a region that is intentionally void as part of the object's geometry. The current binary erosion cannot make this distinction, limiting the applicability of the algorithm to objects that can be fully characterized by their solid geometry alone.

2.4.3 Improvement areas

The analysis of the baseline algorithm's performance and behavior in real-world heritage applications reveals two main areas that require improvement. These two areas translate directly into the algorithm developments presented in this thesis.

The first area deals with the computational performance of the algorithms. The baseline erosion and dilation algorithms have time complexities of $O(n \log n)$ and $O(n^2)$, respectively. Since heritage point clouds often reach tens or hundreds of millions of points, and more computationally expensive operations introduced in this thesis will increase processing time, a faster foundation needs to be created. The main bottleneck for both algorithms is the search and insertion operation within the kD-tree, which grows during dilation (Bentley, 1975a). By replacing this with a spatial hash table hosted on the GPU, the time complexity changes to $O(n)$, which reduces processing time. This idea is the basis of the parallel implementations developed in Section 3.1. The comparison between the baseline and the parallel algorithm will be presented in Chapter 4.

The second area deals with the geometric limitations of the current algorithm when applied to real-life heritage datasets. These limitations can be divided into three sub-problems. First of all, the dilation and erosion do not take density into account. The dilation creates artificial density boundaries and changes the original density of the input data. The erosion, on the other hand, fails to accurately detect objects when the scale of the structuring element does not match the local density of the input. A density-aware implementation will solve these issues and is discussed in Section 3.2. Second, the binary nature of the erosion output poorly handles noisy input data, common in heritage datasets. The implementation of an erosion score creates an output where every point gets assigned an erosion score based on the percentage of points matched from the structuring element, making near-misses more visible. This implementation is further discussed in Section 3.2.2. Third, the structuring element is axis-aligned and therefore cannot detect features that are oriented differently than itself. The correct orientation can either be estimated by calculating the normal based on the surrounding points or by brute-force checking every orientation. These implementations are discussed in Section 3.3.

2.4 Base algorithm review

Together, these enhancements will help make mathematical morphology a practical processing technique for large heritage datasets.

3 Method

Following the improvement areas identified in Section 2.4.3, this chapter presents the developed algorithms for morphological operations on point clouds. For each operation, the changes made to improve the performance are discussed alongside the extensions introduced to account for varying density and surface orientation. In addition to the core morphological operations, a set of helper functions was developed to perform useful operations on point clouds, such as copying attributes and estimating the density. This enables the construction of more complex morphological patterns from simple building blocks. All the developed algorithms are shown in Figure 3.1. Following the algorithm improvements, the application of these methods to heritage point clouds is discussed, with a focus on how their effectiveness is measured across three use cases: segmentation, object detection, and gap filling. Finally, the datasets used for evaluation and their preprocessing steps are described.

This chapter provides a conceptual overview of the implementation. For a detailed technical description of the GPU implementation, shader logic, and data structures, you can find more information in Section A.

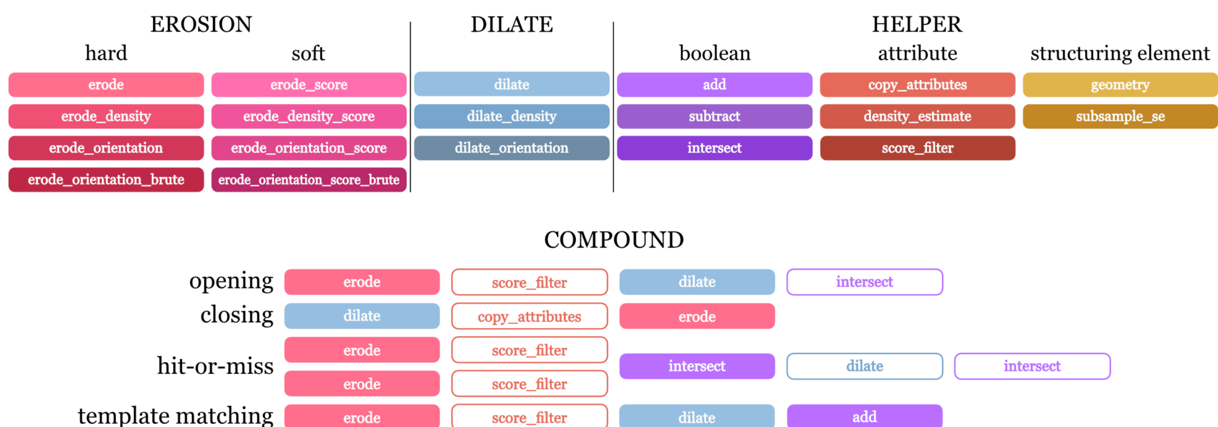


Figure 3.1 *– Schematic overview of the developed algorithms, outlined algorithms are optional in compound operations.*

3.1 Optimization of the base algorithm

3.1.1 Compute Shaders

Shaders are small programs executed in parallel on the GPU, typically used in graphics rendering for effects such as particle simulations or post-processing. Normally, rendering shaders operate on pixels or vertices, but compute shaders are different as they can be applied to any parallel workload, making them more general-purpose. Because modern GPUs contain thousands of processing cores, compute shaders are able to execute the same operation on many elements simultaneously, making them well-suited to operations on large point clouds where the same calculation must be applied independently to each point (Khronos Group, 2012). In this thesis, the OpenGL framework will be used to develop the compute shaders.

Buffers and Uniforms

Since a compute shader runs on the GPU, all data must be transferred from CPU memory to GPU memory before execution. This is done through two types of variables: buffers and uniform variables. A buffer behaves like an array and typically represents the input and output of the algorithm, for example, a buffer of input points and a buffer of output points. The size of each buffer must be declared in advance and allocated on the GPU before the shader runs. Uniform variables only hold a single value that remains constant across all invocations of the shader, such as the cell size of the spatial hash or the minimum point distance threshold. In short, buffers store data for every point, and uniforms store global parameters that dictate the behavior of the entire algorithm.

3.1.2 Spatial Hash Grid

A spatial hash grid is a data structure that maps 3D coordinates to entries in a hash table by first dividing space into a regular, discrete grid and then hashing each grid cell to a table index. Given a cell size c, a point $p = (x, y, z)$ is assigned to the grid cell by flooring $\left\lfloor \frac{p}{c} \right\rfloor = \left(\left\lfloor \frac{x}{c} \right\rfloor, \left\lfloor \frac{y}{c} \right\rfloor, \left\lfloor \frac{z}{c} \right\rfloor \right)$. This cell coordinate is then encoded as a single integer hash value, for example, by computing $c_x \oplus k_1 + c_y \oplus k_2 + c_z \oplus k_3$ using large prime constants k_1, k_2, k_3 to reduce collisions, and the result is mapped into a table with a fixed size (Teschner et al., 2003). When two points map to the same table slot, known as a collision, both are stored in the same bucket. The table must be allocated with sufficient capacity relative to the number of points it will hold to keep buckets shallow and lookups fast.

Time Complexity

The main advantage of a spatial hash grid over a kD-tree is its time complexity. Search, insertion, and deletion all operate in $O(1)$ expected time, granted that the load factor, defined by the point to bucket ratio, does not become too high because of very deep buckets (Teschner et al., 2003). This contrasts with a kD-tree, where the same operations have an $O(\log n)$ average and an $O(n)$ worst-case complexity (Bentley, 1975a). For large point clouds processed in parallel on a GPU, the constant-time of the hash grid is important as it allows thousands of invocations to search the structure simultaneously without the cost scaling with the size of the input.

Nearest Neighbor Search

Finding the nearest neighbor of a point in a spatial hash grid requires searching the cells in the immediate neighborhood of that point, as illustrated in Figure 3.2. If the maximum search radius is known in advance and is set equal to the grid cell size c, then any neighbor within that radius is guaranteed to lie within the $3 \times 3 \times 3 = 27$ cells surrounding the query point's cell. This makes the search bounded and predictable, with no need to expand the search region dynamically (Bentley, 1975b). For the purposes of erosion and dilation, this property is directly applied. Both operations check whether a candidate point lies within a fixed distance of existing points, so the search radius is always known and the 27-cell neighborhood is sufficient.

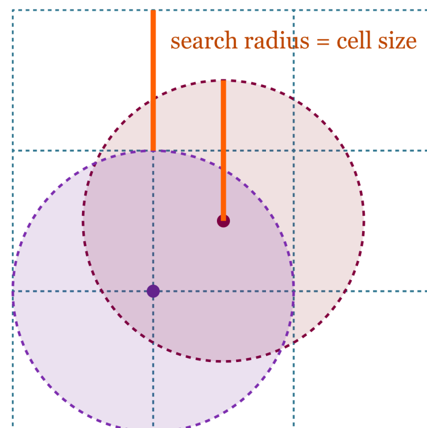


Figure 3.2 *– Ranged nearest neighbor search in a 2D spatial hash grid, red depicts an average search, purple the most extreme case*

When the point cloud has spatially varying density, however, the effective search radius changes across the cloud alongside the local point spacing. In regions of low density, the radius may exceed the cell size, requiring the search to expand beyond the immediate neighborhood into a larger shell of cells. This increases the number of cells that must be checked per query, and the extent of this expansion grows with the ratio between the local search radius and the cell size (Green, 2010). This density-dependent cost is an important consideration for the density-aware variants of the algorithm, discussed in Section 3.2.

3.1.3 Erosion Optimization

Erosion is an embarrassingly parallel operation. This means that every input point can be processed independently without requiring any output data from previous invocations. As a result, there is no possibility for race conditions and memory conflicts to occur, as is the case with dilation. This simplifies the compute shader implementation as each invocation can directly handle one input point.

Before dispatching the erosion shader, all input points are first inserted into the spatial hash grid stored on the GPU, with the cell size equal to the distance threshold as mentioned in Section 3.1.2. This hash is built once, and every point will only reference this grid to check if there are points within its threshold, but never to insert, making it safe for parallel processing.

When running the erosion, three input buffers are uploaded to the GPU: the spatial hash grid, the input points, and the structuring element points, the last two being stored as a flat float vec4 array. The size of these buffers are pre-allocated. The output buffer stores the points that survive the erosion, and an atomic counter stores the number of points.

For every point in the input point buffer, an invocation of the shader will translate the points of the structuring element to the coordinate of the input point and then check if a neighbor exists within the threshold for every translated structuring element point. To find a neighbor, the 27-cell neighborhood in the hash grid is checked with the center cell first, then the 6 face-adjacent cells, followed by the 12 edge-adjacent cells, and ultimately the 8 corner cells. This order is used since cells closer to the center are more likely to contain a neighbor. If any structuring element point fails to find a match, the algorithm will early exit and not write the input point to the output. If all structuring element points find a neighbor, the input point survives and is written to the output using an atomic add.

Finally, the output buffer is read back to the CPU to then be converted back into a CGAL point set. This point set can then be either exported as a PLY file or used as input for another algorithm.

Chunking

Processing large point clouds in a single erosion dispatch might cause the GPU to run out of available memory. To solve this, points can be divided into chunks, which cover a continuous slice of the input buffer. The maximum size of a buffer is inversely proportional to the size of the structuring element, as larger structuring elements require a larger output buffer to be allocated.

The input hash, however, is still built from the complete input point cloud, as this has to remain fixed during the different chunk dispatches. Dividing the hash grid into different chunks would cause points at the edge of chunks to not be able to access all surrounding cells correctly, creating wrong results in those areas.

3.1.4 Dilation optimization

The dilation, on the other hand, is not as easy to parallelize as the erosion. The output of every invocation is used to inform the ones coming after. When processing every input point at once, race conditions might cause unwanted behavior. There is a chance that one invocation writes a point to a grid cell after another invocation has already checked this cell. This will result in the second invocation writing a point to the same cell that does not satisfy the threshold requirement, graphically represented in Figure 3.3. To solve this issue, the input data is partitioned into groups, of which the members are safe to run in parallel. The creation of these groups is discussed further down.

Before running the shader, the input points are inserted into the spatial hash grid stored on the GPU, with the cell size equal to the distance threshold as mentioned in Section 3.1.2. This hash is used to store the output and is updated between every group dispatch, but stays fixed during a single group dispatch.

When running the dilation, every group will dispatch sequentially, running every point within a group in parallel. It takes four buffers as input: The spatial hash grid, the output points, which store the coordinates of the points in the hash, the input points and the structuring element points, the last three are stored as a flat float vec4 array. The size of these buffers are pre-allocated. The candidate point buffer stores the new points that need to be added to the hash between groups, and an atomic counter stores the amount of candidate points.

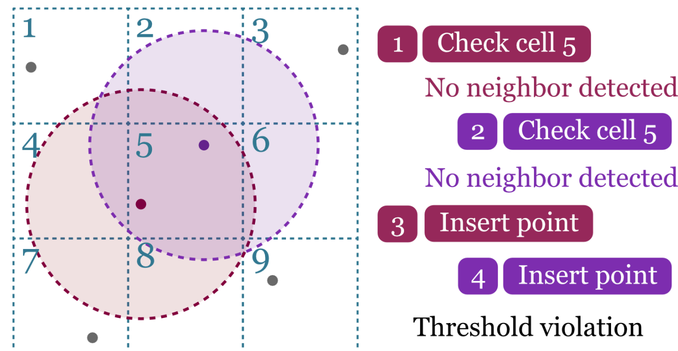


Figure 3.3 *– Graphical representation of race conditions when inserting points in a spatial hash during dilation*

Every invocation of the dilation shader processes one candidate point. A candidate point is a combination of an input point and a single structuring element offset. A 1000-point input and a 10-point structuring element produce 10,000 candidate points. Each invocation checks for the candidate point if there is no point within its threshold yet, atomically adding it to the candidate point buffer if the check succeeds. The 27-cell neighborhood of the candidate point is checked; the order is not important, as all cells need to be checked regardless.

The candidate point buffer is then inserted into the hash and added to the output points, which will be used as input for the next group. Once all groups have been dispatched, the hash is downloaded into a CGAL point set. This point set can be exported as a PLY or be used as input for another algorithm.

Grouping

Partitioning the input data is a common strategy to solve parallel memory conflicts like race conditions. Figure 3.4 shows how the points are grouped such that no two points within the same group fall within the diameter of the structuring element of each other. This way, every point in a group — and their candidates — can be processed in parallel, as they can never overlap. The groups are then processed in sequence, with one dispatch per group.

A greedy algorithm assigns each point to a group, using a hash grid scaled to the size of the structuring element. For every point, the algorithm checks the 27-cell neighborhood and finds the first group that is not yet present and assigns it to that point. The resulting groups are then ordered from largest to smallest. While this greedy approach does not minimize the total number of groups, it is efficient to compute and produces partitions that are well-suited to GPU dispatch in practice.

One caveat is that as the physical diameter of the structuring element increases, it takes longer to assign the groups as more points need to be checked per step. This also results in more groups being generated. The effects of this on the total processing time are discussed in Section 4.1.2

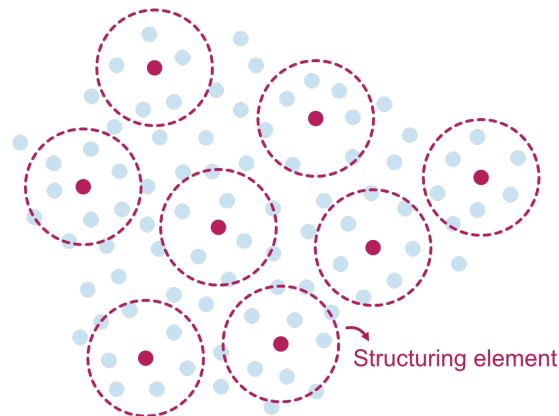


Figure 3.4 – *Red points of which the structuring element extents do not overlap belong to one group*

3.2 Density-aware operations

3.2.1 Self-adaptive structuring elements

When applying mathematical morphology at varying densities, you need a structuring element that can store the density level at which a structuring element point offset should be used. Two ways of creating such self-adaptive structuring elements are implemented, both having their own advantages and shortcomings.

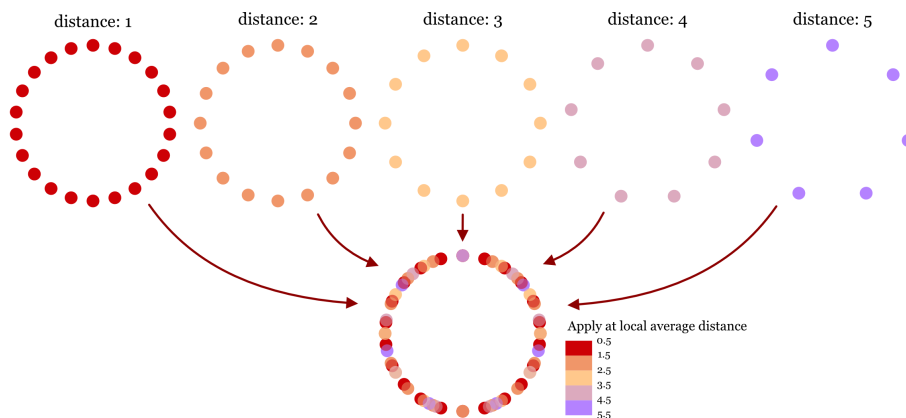


Figure 3.5 – *Self-adaptive structuring element generated from a circle by discretely subsampling at different minimum distances*

The first approach creates a separate structuring element for every discrete density increment by sampling points evenly at the required distance for that density level. This process can be performed using software like CloudCompare or directly within the CGAL library. These individual structuring elements are then merged into a single combined structuring element, with each point labelled by the density increment it belongs to. This value is stored in the fourth value of the point vector. When processing a point in the shader, only the subset of points for

which the density matches the local density of the input point is used. This approach makes sure the structuring element is equally spaced at each density level, since each individual structuring element is subsampled independently. The drawback is that the total number of points grows with the number of density increments required. This increases the amount of data uploaded to the GPU and requires more checks when running an algorithm.

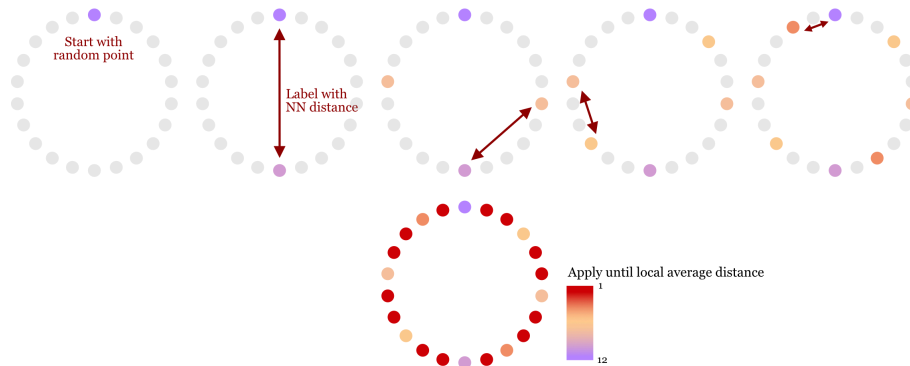


Figure 3.6 *– Self-adaptive structuring element generated from a circle by continuously assigning distances to points.*

The second approach assigns a density label to each point in a single structuring element. The label reflects the minimum local density at which that point should be used instead of the increment band. The element is constructed using a greedy algorithm. A starting point is selected, then the point furthest from it is added and labelled with the distance to its nearest neighbor. This process repeats, at each step, the point furthest from all currently accepted points is added and labelled, until all points of the structuring element are labelled. The result is a single set of points that becomes coarser as the local density decreases. At high densities, all points are active, while at lower densities, only the coarser, more widely spaced points are used. Because the structuring element stores the original structuring elements with continuous labels, no extra points have to be uploaded to the GPU. However, the greedy construction algorithm does not create equally spaced points at each density level in the way that the first approach does.

3.2.2 Erosion score

Eroding using an erosion score presents a version with a softer continuous output, rather than the hard binary output of the regular erode. Rather than discarding a point when it fails the erosion, it now gets attributed a score based on how well the structuring element matches at that location. A score of 1 indicates a perfect match, while a score of 0 means no neighbors were found for any of the translated structuring element points. An intermediate score represents a partial match; the useful cut-off depends on the input data, structuring element, and threshold size, and is therefore hard to predict beforehand. This way, near-misses can be more easily detected, and more information can be obtained from the erosion.

To implement the erosion score in the shader, some changes had to be made. Instead of allowing the algorithm to early exit when any structuring element point fails the neighbor check, the score version needs to evaluate all points of the structuring element to calculate the correct

percentage of matches. The way the output score is calculated is illustrated in Figure 3.7. This change does increase the workload per invocation, as the number of checks is higher due to the lack of early exit.

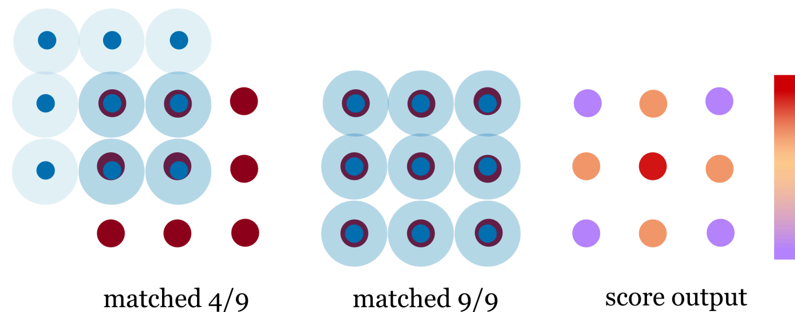


Figure 3.7 *– Graphical representation of the erosion score algorithm, the input (red) gets assigned a score by dividing the matches by the total number of points in the SE (blue)*

Since the output buffer is already a vec4 array with an unused w coordinate, the calculated score can be stored directly in the output buffer. This eliminates the need for a separate score buffer to be read back to the CPU. Afterwards, the score is stored as a CGAL property map for exporting as a scalar field or for later use.

3.2.3 Density-aware erosion

Standard erosion fails to accurately erode on point clouds with a varying density. A single minimum distance threshold set to erode at a high density will fail to find matches in low-density areas, while the reverse causes false matches in high-density areas. A density-aware variant of the erosion dynamically adapts the spacing of the structuring element to perform well at different densities, and it also matches the threshold to the local density of the point cloud. This combination will make the erosion perform more consistently across different densities.

Before the shader is run, the local density is estimated for every input point using a density estimator shader implemented on the GPU. It finds the 5 nearest points to every input point and calculates the average distance to insert as the density value. Since a hash grid is used here as well, it is important to note that a non-ranged nearest neighbor search is not guaranteed to be found within a certain cell neighborhood. To account for this, a maximum shell is set to end the search early if 5 points cannot be found in those shells. The distance value is stored on the w coordinate of the vec4 output buffer and then added as a density property map to the CGAL point set.

During a shader invocation, before checking any translated structuring element point, the shader checks the density value of the input point to calculate the number of shells to search for nearest neighbors. A point in a high-density area might only need a 27-cell neighborhood, while in a low-density area, 64, 125, or more cells need to be checked. Every structuring element point also has its own density label. The shader will check if the point needs to be considered for the erosion. In the case of discrete density increments, it will check if the point belongs to the

right increment, while for the continuous subsampled implementation, it checks if the density of the structuring element is above that of the input point.

The calculation of the score is also adapted to only take the relevant point into account. This means that points in lower-density areas will use fewer points to calculate their score and therefore can reach a high score more easily, while in higher-density areas, the opposite is true.

3.2.4 Density-aware dilation

The density-aware dilation similarly first calculates an average distance for the input points before running the shader. This estimated density is stored on the w coordinate of the vec4 input buffer. This local density of the input, combined with the density value of the structuring element points, determines what points of the structuring element should be applied for the dilation.

When the shader checks if a candidate point is far enough away from any point already present in the output, the density-aware variant uses the local density as a threshold instead of a fixed global value. New points are inserted to match the spacing around the input point. Even if the density of the structuring element is lower than the one at the input point, new points can still be added from other input point invocations to increase the density around that point. This makes it so the local density is preserved everywhere in the dilated point cloud.

3.3 Orientation-aware operations

In heritage point clouds, objects are free to rotate on the xy-plane, meaning that the same object might be presented at different orientations. The current implementation of the morphological operations makes use of an axis-aligned structuring element that is always applied at the same orientation. This means that features that are oriented differently will not be correctly detected or filled. By allowing the structuring element to rotate to best fit the input point cloud, the operations will become more generally applicable. The rotation of the structuring element is implemented using a TNB matrix, which remaps the local coordinate frame of each point.

3.3.1 TNB matrix rotation

Normally, when rotation object in 3D space, there are three separate axes of rotation, with their own rotation matrix, which leaves too much freedom to determine the exact rotation with just one point and an orientation vector. However, another strategy for rotation is remapping the coordinate axes of the structuring element to fit a local coordinate system at each point. This approach is conceptually similar to UV texture mapping in computer graphics.

The local coordinate system of a point is defined by three orthogonal vectors: the normal N, the tangent T, and the bitangent B. The normal of the point is set equal to the y-axis of the structuring element. When using UV texture mapping, this is normally the z-axis, so a TNB matrix is created here instead of the usual TBN matrix. Taking into account the handedness of the final matrix, the tangent is calculated as the cross product of the normal and the z-axis, or the x-axis in case the normal is equal to the z-axis. The bitangent is then derived as the cross

product of the tangent and the normal, creating three orthogonal axes (de Vries, 2020). The resulting TNB matrix is defined in Equation (3.1):

$$
\begin{aligned}
\boldsymbol{T} &= \boldsymbol{N} \times \|\hat{\boldsymbol{z}}\| \\
\boldsymbol{B} &= \boldsymbol{T} \times \boldsymbol{N} \\
\boldsymbol{M}_{\text{TNB}} &= \begin{pmatrix} T_x & N_x & B_x \\ T_y & N_y & B_y \\ T_z & N_z & B_z \end{pmatrix}
\end{aligned}
\tag{3.1}
$$

The resulting orientation of an arrow structuring element on the xz-plane pointing in the direction of the z-axis after being transformed by the TNB matrix is shown in Figure 3.8. The tangent direction can be chosen freely, in this case, to maintain the upwards direction of structuring elements.

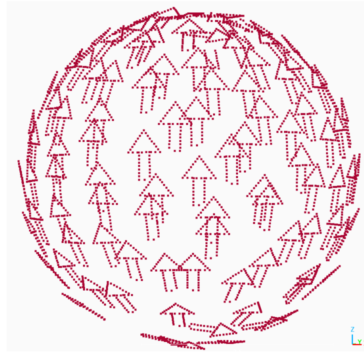


Figure 3.8 *– Direction of the rotation for an arrow on the xz-plane pointing in the direction of the z-axis*

3.3.2 Orientation-aware erosion

To find the correct orientation of the structuring element, two approaches can be used. A first option is to estimate the correct orientation beforehand, for example, by estimating the normal in a program like CloudCompare and then rotating the structuring element accordingly using the TNB matrix. However, this approach might not always be suitable. The algorithm assumes that the normal of the reference point and the y-axis of the structuring element align perfectly. This requires that the surface normals at the reference points are consistent and smooth enough that the TNB rotation is stable. In areas with high geometric variance, due to noise or complexity, like ornamental details or rough surface textures, normals may be calculated more erratically between neighboring points, causing the erosion to fail. A second option, therefore, is to brute force different possible orientations and select the best match. This approach is computationally more expensive, but does not suffer from the issues mentioned before.

Erosion with predetermined orientation

When a surface normal is available or an estimation can be made, the shader implementation is quite straightforward. Before translating the structuring element to the reference point, a TNB matrix is constructed from the point normal, and the structuring element is rotated accordingly. The rest of the shader then operated the same. The normal must be uploaded as a separate

input buffer alongside the point positions since a vec4 can only carry four values. Since xyz are used for position, this leaves no room for the normal in the same buffer. Similarly, an output normal buffer is also necessary to store the normal of surviving points for further operations.

Erosion with unknown orientation

When no reliable normal estimation exists or where the exact orientation of a target object is unknown, a brute-force approach might be preferred. Rather than relying on a single TNB matrix rotation, the shader sweeps the structuring element in all possible orientations over the reference point. During the hard erosion, the first orientation to fully survive is selected, while for the soft version, using the erosion score, the orientation with the highest score is selected.

The orientations used in the sweep are defined using spherical coordinates θ and φ as shown in Figure 3.9, where θ controls rotation around the vertical axis and φ controls inclination from the vertical. If the target feature is known to lie roughly on the xy-plane, the search can be restricted to rotations around the z-axis only (varying θ with φ fixed at $\pi/2$), reducing the search space.

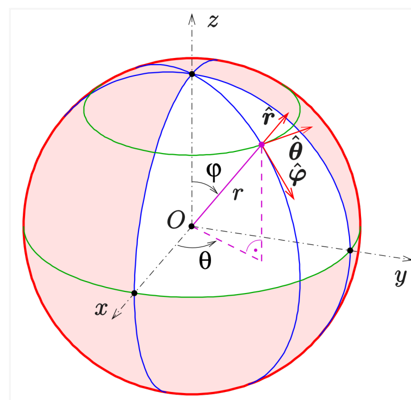


Figure 3.9 *– Spherical coordinate system with local basis vectors, source: (Ag2gaeh, 2008)*

To keep the computation from exploding in size, the search is performed in two phases: a coarse sweep using a low-density structuring element to find the best direction. This is then followed by an erosion using the high-density structuring element in that direction. This two-step approach lowers the precision of the search, but is able to heuristically find a suitable orientation faster.

3.3.3 Orientation-aware dilation

The orientation-aware dilation is only implemented for the scenario in which a predetermined orientation is provided. By rotating the structuring element using the TNB matrix to align it with the local surface orientation, gap filling can be performed on many differently angled surfaces at the same time, rather than only in the axis-aligned direction of the structuring element. Similarly to the erosion, a separate normal input buffer needs to be uploaded to the GPU.

The brute-force orientation search is not applied to dilation because the operation needs to be fully completed before any further erosion can be applied. Furthermore, the current grouping

strategy makes it so there is no practical way to simultaneously perform an orientation search with the incremental candidate acceptance that the dilation requires. The primary use for orientation-aware dilation is the closing operation, in which dilation is followed by erosion to fill holes while preserving overall shape. Since a brute-force closing implementation would not result in any significant performance gains, it has not been implemented.

3.4 Helper and compound operations

The algorithms for opening and closing are powerful, but have limited applications for heritage point clouds when used only by themselves. To neatly chain these base morphological algorithms into each other to form compound operations, such as the opening used for object detection, helper operations need to be created. There are three types of these helper operations: boolean operations, structuring element modification, and attribute handling. Furthermore, these helper operations can be used to calculate metrics, such as the number of correct detections, to evaluate the algorithm output.

Four compound operations will be discussed: closing, opening, hit-or-miss transform, and template matching. The closing involves a dilation followed by an erosion and is mainly used for gap-filling purposes. The opening is obtained by first eroding and then dilating. Its main applications are segmentation and object detection. Third, the hit-or-miss transform requires the intersection of two separate erosions with complementary structuring elements. It is also used for segmentation and object detection, as well as edge detection. Finally, the template matching involves the detection of a partial object followed by the dilation with the full object, and is used for gap-filling. While the development of these compound operations is not the focus of this thesis, and they will not be further evaluated in the results, the helper operations that make them possible will be described here for completeness.

3.4.1 Boolean operations

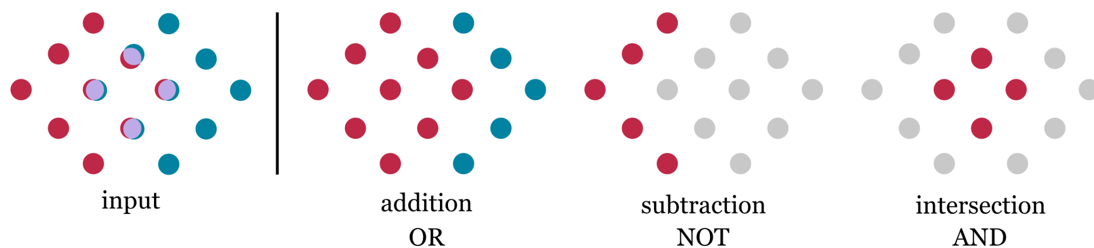


Figure 3.10 *– Graphical representation of boolean operations applied to point clouds*

Boolean operations are logical set operations that combine inputs, like geometries, with processes like AND, OR, and NOT. To translate these processes to point clouds, the overlap is defined using a threshold distance, consistent with the morphological operations. However, this design decision causes some of these operations to not be fully commutative, unlike their mathematical counterparts, since the exact coordinates will differ based on the order of inputs. In Figure 3.10, the three implemented boolean operations: addition, subtraction, and intersection, are visually represented.

Addition

The addition algorithm is the equivalent of the logical OR. The resulting output will contain points from both point clouds, with duplicates being removed. Points from the first point cloud are given priority over the second.

The algorithm first uploads all points of the first point cloud to a spatial hash grid. Then a shader is used to check in parallel for every point of the second point cloud if they can find a neighbor within the threshold distance in the hash. Points that fail this check are added to the output buffer. The output buffer is then added to the hash before being output to a CGAL point set. In essence, this algorithm is a simplified version of the dilation, where the points from the second point cloud can be seen as a single group of candidate points, instead of being generated from every input and structuring element point pair. A similar result can also be obtained using the *merge clouds* and *remove duplicate points* tools in CloudCompare, although this does not guarantee all the points of the first cloud are preserved.

Addition can be used for creating structuring elements by combining simple base shapes, for adding points obtained from template matching, and for merging outputs obtained from operations using different structuring elements, such as edge detection in multiple directions.

Subtraction

The subtraction algorithm is the equivalent of the logical NOT. The resulting output consists of the points of the first point cloud that could not find a neighbor within the threshold distance in the second point cloud.

Figure 3.11 – *Approximation of the surface-based dilation obtained by subtracting two dilations using spherical SEs one threshold distance apart in radius.*

The algorithm first uploads all points of the second point cloud to a spatial hash grid. Then, for every point of the first point cloud, a shader is used to check if there is a point in the hash within the threshold distance. Points that fail this check are added to the output buffer and output as a CGAL point set. This algorithm is similar to the addition, but does not add the points of the first and second point clouds together. Subtraction is not commutative, as only the points of the first point cloud are maintained.

Subtraction is used for the segmentation of a point cloud to remove points from the input between openings, for computing false positives and false negatives to evaluate the algorithms, and for generating negative structuring elements used in the hit-or-miss transform.

Another use case for the subtraction is the approximation of the surface-based dilation and erosion. By dilating twice with structuring elements that differ by only one threshold distance in radius and then subtracting the two outputs, an outer shell can be obtained as shown in Figure 3.11. With a reliable normal estimation or by using the LiDAR scanning angle, available in some datasets, it is possible to separate the inner and outer shell, respectively, the surface-based erosion and dilation. This concept, however, is not further explored in this thesis, but could provide the basis for the skeletonization, simplification, and denoising of point clouds (Calderon & Boubekeur, 2014).

Intersection

The intersection algorithm is the equivalent of the logical AND. The resulting output consists of the points of the first point cloud that were able to find a neighbor within the threshold distance in the second point cloud.

The algorithm first uploads all points of the second point cloud to a spatial hash grid. Then, for every point of the first point cloud, a shader is used to check if there is a point in the hash within the threshold distance. Points that succeed this check are added to the output buffer and output as a CGAL point set. This algorithm is the complement of subtraction. Intersection is usually commutative, but the point cloud implementation causes the intersection to only contain the exact point coordinates of the first point cloud.

Intersection is used for the opening to return to the original input after the erosion and dilation, for the hit-or-miss transform to obtain the final output from the intersection of the two erosions, and for the calculation of the true positives to evaluate the algorithms.

3.4.2 Structuring element modification

A large factor influencing the output of morphological operations is the definition of the structuring element. Structuring elements consisting of simple geometries can be easily generated to match the density of the input data. Furthermore, for the density-aware algorithms, the structuring element needs to have a density property to correctly adapt to the local density as specified in Section 3.2. To facilitate these two requirements, dedicated helper functions were developed.

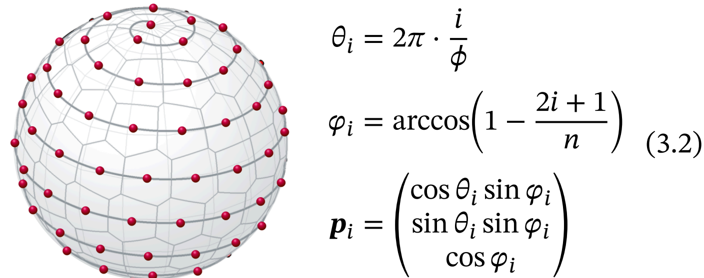
Geometry generation

Most basic geometries used as structuring elements, like lines, planes, and cubes, are generated using simple parameters, such as point spacing and extent. This allows the geometry to be matched to the density and scale of the input point cloud.

However, spheres present a particular challenge, since distributing points uniformly across a spherical surface is a non-trivial problem. The paper by Keinert et al. (2015) presents an effective solution using a Fibonacci spiral defined by the golden ratio. This spiral wraps around the

sphere in such a way that points distributed along its trajectory have a near-uniform spacing. The resulting distribution is reminiscent of the spiralling pattern observed in the heads of sunflowers.

Using the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$, the i-th point, where $i = 0, 1, \ldots, n-1$, is defined as:



$$\theta_i = 2\pi \cdot \frac{i}{\phi}$$

$$\varphi_i = \arccos\left(1 - \frac{2i+1}{n}\right) \qquad (3.2)$$

$$\boldsymbol{p}_i = \begin{pmatrix} \cos\theta_i \sin\varphi_i \\ \sin\theta_i \sin\varphi_i \\ \cos\varphi_i \end{pmatrix}$$

Figure 3.12 *– Points arranged on a sphere using the Fibonacci spiral, image source: (cchu79, 2023)*

These simple geometries can then be used to further generate structuring elements. For example, the negative window structuring element used furing the hit-or-miss transform was generating by dilating the window with a plane parallel to its surface and then subtracting the original window points from the dilation result.

Self-adaptive structuring element generation

The density-aware erosion and dilation require a structuring element with a density attribute specifying when each point should be considered, as described in Section 3.2. Two implementations are provided: a discrete and a continuous approach.

The discrete implementation subsamples the structuring element at predefined point spacings using the CGAL *wlop_simplify_and_regularize_point_set()* function. Upsampling is also supported through the CGAL library, though it is rarely necessary in practice since structuring elements denser than the input do not provide any additional benefits. The subsampled elements at each density level are combined into a single CGAL point set with a density property map specifying the density band at which each point should be applied.

The continuous implementation begins with the point of the structuring element closest to the origin. The distances from all remaining points to this initial point are calculated, and the point with the greatest distance is selected next. This point receives a density label equal to that distance. The initial point is assigned the same density label. All remaining points then calculate their distance to the newly added point, updating their stored distance if this value is smaller than the previously recorded one. This process repeats by selecting the point with the highest stored distance and updating the distances of remaining points until all points have been assigned a density label. The result is a CGAL point set with a density property map that stores a continuous ordering of points.

3.4.3 Attribute handling

Point clouds typically contain more attributes than just coordinates, such as normals, color values, and scalar fields. These attributes must be preserved during morphological operations. Additionally, the density-aware and orientation-aware variants require a density attribute and surface normals, respectively, as inputs. These both need to be computed if they are not already present in the input data. The erosion score output is also stored as a point attribute and must be filtered before being passed to the following operations during compound operations, such as opening or hit-or-miss.

Attribute interpolation

When a single input point wants to dilate to a candidate location, calculating the attributes is as simple as copying them directly. However, when multiple input points want to dilate to the same candidate location, a conflict occurs as to which attribute values should be copied. Furthermore, during the opening, the initial erosion deletes points from the cloud. This causes a loss in attribute data that cannot be recovered from the erosion alone. Two strategies exist to fix this issue: interpolating attribute values during the shader or recovering them from the original dataset after the operation, which was previously proposed by Balado et al. (2021).

The density and normal attributes used by the density-aware and orientation-aware variants are maintained during their respective algorithms using a first-come, first-served implementation. This is acceptable for the erosion, where no new points are introduced. However, this becomes problematic during the dilation, where new points are created where attributes are copied from the point that was processed first, instead of a point that might be physically closer. As a result, density and normal values are typically recalculated or copied from the original input after dilation. For other attributes such as color or intensity, CloudCompare provides built-in interpolation functions.

Density and normal estimation

The local point cloud density is estimated, for each point in parallel, by taking the five closest neighbors and calculating the average distance. This approximation, also employed in the baseline algorithms, produces reliable estimates under normal conditions. An important edge case is when a point cloud contains five or more duplicates of a single point. In this case, they will all have a density value of 0, which will result in unexpected behavior as the minimum distance threshold is determined by this value. Because of this, duplicate points are removed before the density estimation. Other density estimations can also be used, as long as the density value represents the distance between points. If a point cloud is given as input without a density attribute, it will automatically be calculated before the shader is dispatched.

The normals are estimated in CloudCompare using the *Hough normal estimation* plug-in tool, which gives reliable results near sharp edges like corners. Any other normal estimation can also be substituted. If an input without normals is given, the orientation-aware algorithms will default to regular dilation or erosion.

Score filtering

The erosion score attribute can be filtered by looking at every point and checking if its value survives the filter. For opening operations, the filter retains points above a specified threshold, while for the negative erosion step of the hit-or-miss transform, points below the threshold are retained instead. Score filtering can also be applied to separate class labels for calculating segmentation metrics. This operation can be performed within CGAL, in CloudCompare, or using Python libraries such as Plyfile or Laspy.

3.4.4 Compound operations

Compound morphological operations form the backbone of the heritage applications evaluated in this thesis. Instead of being implemented as standalone algorithms, they are constructed from the previously developed algorithms, erosion, dilation, and the Boolean helpers, as building blocks. Some steps in the compound operations are optional depending on the specific application.

Opening

Opening consists of an erosion followed by a dilation, optionally including score filtering and an intersection with the original point cloud:

$$\text{erosion} \rightarrow \text{(score filter)} \rightarrow \text{dilation} \rightarrow \text{(intersection)} \tag{3.3}$$

The erosion first removes points that do not match the structuring element. Oftentimes, a binary erosion output fails to detect all objects in a noisy point cloud, and so, if the erosion score variant is used, these scores need to be filtered before moving on to the dilation. The dilation then restores the spatial extent of the surviving points. The output is then all the structuring element matches the algorithm could find. In many detection use cases, the output should match the exact coordinates of the input. An optional intersection can be used to achieve this, which simultaneously reduces outlier noise.

Closing

Closing consists of a dilation followed by an erosion, with an optional intermediate step to restore or generate density and normal attributes lost during dilation:

$$\text{dilation} \rightarrow \text{(attribute recovery)} \rightarrow \text{erosion} \tag{3.4}$$

The dilation first fills in gaps in the input point cloud, adding points to the empty areas. Since the dilation can result in an unreliable attribute interpolation, the normals and density values can optionally be recovered from the input before moving on to the erosion. The erosion then removes all points that do not match the structuring element, maintaining only the actual filled gaps.

Hit-or-miss transform

The hit-or-miss transform involves the intersection of two erosions with complementary structuring elements, followed by a dilation. Optionally, it includes score filtering and an intersection with the original point cloud:

$$\left.\begin{array}{l}\text{erosion}^{+} \rightarrow \text{(score filter)} \\ \text{erosion}^{-} \rightarrow \text{(score filter)}\end{array}\right\} \rightarrow \text{intersection} \rightarrow \text{(dilation)} \rightarrow \text{(intersection)} \tag{3.5}$$

The positive erosion identifies all points where the structuring element geometry is present, while the negative erosion verifies that the negative space around the structuring element matches as well. Notably, for the negative erosion, the algorithm checks if there is no neighbor for every point in the structuring element, using opposite logic to the regular erosion. If the score variant is used, these need to be filtered before moving on to the intersection. The intersection then overlays the positive and negative erosion and only keeps the points that satisfy both. For object detection, a further dilation and optional intersection are necessary to return all points belonging to the structuring element geometry, although for uses like edge detection, this is optional.

Template matching
Template matching uses a partial structuring element for detection and a complete structuring element for reconstruction. The resulting output is then added back to the original input:

$$\text{erosion}^{p} \rightarrow \text{(score filter)} \rightarrow \text{dilation}^{c} \rightarrow \text{addition} \tag{3.6}$$

The initial erosion with a partial structuring element, usually the most distinctive area of the target object, finds all objects that fit that geometry. If the score variant is used, these scores need to be filtered before moving on to the dilation. The dilation happens with a structuring element of the complete object, filling in the parts that the partial objects miss. The resulting output is then added to the original input point cloud to fill in these missing regions.

3.5 Heritage applications

To assess the usefulness of the developed algorithms, they will be tested on various heritage use cases, particularly: segmentation, object detection, and gap-filling. The visual representation of these applications is shown in Figure 3.13.

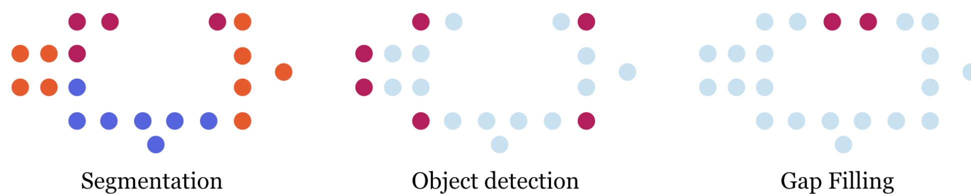


Figure 3.13 – *Graphical representation of the applications of mathematical morphology*

3.5.1 Segmentation

Point cloud segmentation assigns a semantic label to each point, dividing the cloud into meaningful categories, in this particular case, for architectural features found in heritage buildings. Segmentation is performed here using mathematical morphology, specifically the opening and hit-or-miss transform. It can detect regions of the point cloud that match the geometry of the structuring element representing a specific category. For example, a horizontal plane can be applied to detect floors, while a cylinder is better for columns. Where the geometric nature of

a category requires it, orientation-aware operations can be used, either by aligning the structuring element with the estimated surface normals or through a brute-force orientation search, so that detection is not limited to a single fixed direction.

Process
The segmentation works sequentially through a fixed order of categories. Each category uses one or two structuring elements to find all points belonging to that category. Once these points have been identified, they are removed from the input for the next category. This sequential process makes sure that points get assigned at most one label, but also means that the segmentation result depends on the order of the categories. The structuring elements can be tailored to the individual point cloud, for example, by sampling a column or arch from the input directly, but each category is limited to at most two elements to keep the process workable and to make the geometric assumptions explicit.

The order in which categories are processed is based on how distinctive their geometries are and, therefore, how easy they can be identified without confusion with other categories. The most distinct categories are processed first. Columns are processed at the beginning since their cylindrical shape is easy to isolate. Similarly, arches, stairs, and vaults can be detected with relatively few false positives. Walls, on the other hand, are left until late in the order, since they are broadly defined as any vertical surface and share geometric properties with more specific categories like doors and windows. Removing doors and windows first reduces the risk of false detections. Any point for which no category is detected by the end of the process is assigned to the left-over class, labelled as 9 (other).

It is worth noting that the goal here is not to produce a generic, broadly applicable segmentation algorithm. Mathematical morphology is inherently data specific, as the structuring elements are tailored to the target objects in a point cloud. The aim is instead to assess how well mathematical morphology can segment an individual heritage point cloud under controlled conditions, which is a question of feasibility rather than generalizability.

Evaluation
The final segmentation is evaluated using a set of standard metrics. The main metric is Mean Intersection over Union (mIoU), which divides the number of correctly assigned points for each class by the total number of points either predicted to be in that class. Then it averages this for all classes. Overall Accuracy (OA) gives the percentage of all points that are correctly labelled, although it can be misleading when the categories are unbalanced, for example, when floors make up half of a point cloud. Mean Precision (mPrec) and Mean Recall (mRec) are used to measure the purity and completeness of each predicted class. Precision measures how reliably a predicted label is correct, and recall measures how completely a ground truth category is found. The Mean F1 (mF1) balances both these values. These are averaged per class and are therefore more robust for class imbalance. Besides these quantitative metrics, the labelled point clouds are inspected visually, since metrics alone do not always reveal errors like structuring element artefacts.

Let C be the number of classes, N the total number of points, and TP_i, FP_i, FN_i the true positives, false positives, and false negatives for class i. The metrics are defined as:

$$\mathrm{mIoU} = \frac{1}{C}\sum_{i=1}^{C}\frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FP}_i + \mathrm{FN}_i} \quad \mathrm{OA} = \frac{\sum_{i=1}^{C}\mathrm{TP}_i}{N}$$
$$\mathrm{mPrec} = \frac{1}{C}\sum_{i=1}^{C}\frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FP}_i} \quad \mathrm{mRec} = \frac{1}{C}\sum_{i=1}^{C}\frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FN}_i} \quad \mathrm{mF1} = \frac{1}{C}\sum_{i=1}^{C}\frac{2\cdot\mathrm{TP}_i}{2\cdot\mathrm{TP}_i + \mathrm{FP}_i + \mathrm{FN}_i} \tag{3.7}$$

Datasets

The method is tested on the Images&PointClouds Cultural Heritage Dataset, further described in Section 3.6.3, which provides ground truth semantic labels for five heritage point clouds, obtained with terrestrial photogrammetry. The featured heritage sites are located in Florence, Italy.

3.5.2 Object detection

Where segmentation assigns a label to every point in the cloud, object detection has a narrower goal, namely, finding all instances of a specific object within the point cloud. This is done using either the opening or the hit-or-miss transform with a structuring element representing that object. The structuring element can be the full object or a subsampled version of it, as long as it captures the geometry needed for detection.

Process

Each object is detected by applying the hit-or-miss transform or opening with one of the orientation or density-aware algorithm variants described in the previous sections, depending on what is most appropriate for the object in question. For the hit-or-miss transform, a negative structuring element is also required. This is generated by dilating the original structuring element with simple geometric shapes and then subtracting the original points. For example, a shell of points just outside the object boundary that the target region must be free of can be used to separate an object from noisy regions like foliage.

Evaluation

Unlike segmentation, which is evaluated per point, object detection is evaluated per object instance. The main metrics are Recall, which measures the percentage of objects in the scene that were successfully detected, and Precision, which measures the percentage of detections that correspond to an actual object rather than a false positive. These two are combined into an F1-score, which gives a single balanced measure of detection quality. The results are also inspected visually.

$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \quad \mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \quad \mathrm{F1} = 2 \times \frac{\mathrm{Precision} \times \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}} \tag{3.8}$$

Datasets

Object detection is tested on two datasets from OpenHeritage3D, further described in Section 3.6.2, both acquired with terrestrial LiDAR. The first is the Fontana dei Mesi in Turin, Italy,

where a pillar is used as the target object for a regular hit-or-miss transform, and a decorative shell for a brute-force opening. The second is Houghton Hall in Norfolk, UK, where a window is detected using an orientation-aware hit-or-miss transform, and an acorn ornament using a density-aware opening.

3.5.3 Gap filling

Gap filling addresses a common problem in heritage point clouds: regions where the surface is missing due to occlusion, scanner limitations, or physical inaccessibility. Two steps are identified: edge detection and surface reconstruction. Edge detection locates where the point cloud is unexpectedly interrupted, while surface reconstruction fills in the missing geometry.

Process

Edge detection is performed using the hit-or-miss transform. The positive structuring element checks whether a point lies on the surface, while the negative element checks whether that surface is interrupted at that location, as illustrated in Figure 3.14. The underlying assumption is that the point cloud describes a watertight object, so any interruption in the surface can be treated as a gap rather than an intended boundary.

Surface reconstruction is based on closing, which fills gaps by dilating the point cloud into the missing region and then eroding back to preserve the overall shape. Lines and planes are used as the primary structuring elements since they are the most generic and perform well across a range of gap types. Other structuring element shapes could, in principle, be used to address more specific problems, like corners. A second reconstruction strategy is object reconstruction using template matching, where a partial object is eroded to find matching locations in the point cloud, and then dilated with a complete reconstruction to fill in the missing geometry. The result is added back to the point cloud rather than extracted from it, making it complementary to the object detection use case described earlier.

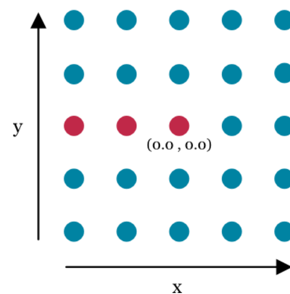


Figure 3.14 *– Structuring element used for edge detection, points used in the positive erosion (red) complement those used in the negative (blue)*

Evaluation

Gap filling is evaluated using the Chamfer Distance, defined between point sets P and Q in Equation (3.9), and measures the average distance between the filled point cloud and the ground truth surface, and is the standard metric for assessing geometric reconstruction quality. Precision, Recall, and F1-score, previously defined in Equation (3.8) are also reported for both

gap filling and edge detection, measuring how accurately gaps are located and how completely they are filled. Template matching is not evaluated in depth here since its detection step is similar to the object detection experiments described in the previous section.

$$\mathrm{CD}(P, Q) = \frac{1}{|P|} \sum_{p \in P} \min_{q \in Q} \|p - q\|^2 + \frac{1}{|Q|} \sum_{q \in Q} \min_{p \in P} \|q - p\|^2 \tag{3.9}$$

Datasets

The experiments use the OpenHeritage3D dataset of the Palacio de Bellas Artes in Mexico City, Mexico, further described in Section 3.6.2. A terrestrial photogrammetry point cloud serves as the ground truth. To simulate realistic occlusion patterns, a hidden point filter is applied to the ground truth to artificially mimic a LiDAR scan, removing points that would not be visible from a given scanner position and creating controlled gaps for the algorithm to fill.

3.6 Datasets

To test the algorithms in a real-world heritage context, several datasets were selected to apply the operations to. The basic details of these datasets are described here; for more information about the preprocessing, the reader is referred to Section A

3.6.1 Stanford 3D Scanning Repository

The Stanford 3D Scanning Repository provides two well-known test models: the Armadillo and the Bunny, which are used here for development and code testing. Both are standard benchmarks in computer graphics and provide a convenient way to verify algorithm correctness before moving to more complex heritage data. The point clouds of both models are shown in Figure 3.15.

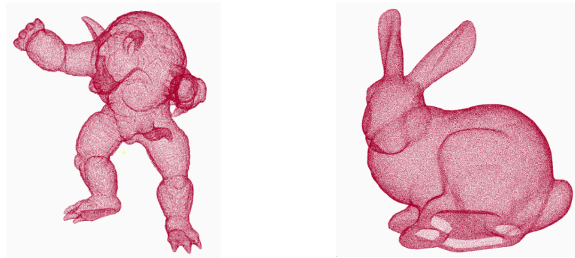


Figure 3.15 *– Stanford Armadillo and Bunny point clouds*

3.6.2 OpenHeritage 3D

The OpenHeritage3D datasets used in this work consist of three heritage point clouds, shown in Figure 3.16, with statistics summarized in Table 1. The Fontana dei Mesi (FM) in Turin, Italy, (Teppati Lose et al., 2023), and Houghton Hall (HH) in Norfolk, United Kingdom, (Kress Foundation, 2021) are both terrestrial LiDAR acquisitions that have been merged and subsampled to a uniform density of 2 cm. These two datasets are used for object detection. In addition, a density-banded variant of the Houghton Hall dataset is prepared by cutting the point cloud into five horizontal slices, each subsampled at a decreasing density ranging from 2 to 10 cm, to test the density-aware algorithm variants.

***Figure 3.16** – Datasets from OpenHeritage3D, source: (CyArk, 2019; Kress Foundation, 2021; Teppati Lose et al., 2023)*

The Palacio de Bellas Artes (PB) in Mexico City, Mexico (CyArk, 2019) is a photogrammetry reconstruction that has been cleaned and subsampled. A hidden point filter is then applied from multiple angles to simulate LiDAR occlusion, producing a set of controlled gaps for the gap-filling experiments.

***Table 1** – Information for the OpenHeritage3D Datasets*

Name	Code	Points	Source	Density
Fontana dei Mesi	FM	13,943,961	Terrestrial LiDAR	0.02 m
Houghton Hall	HH	5,431,780	Terrestrial LiDAR	0.02 m
Palacio de Bellas Artes	PB	2,768,939	Aerial + Terrestrial Photogrammetry	0.05 m

3.6.3 Images&PointClouds Cultural Heritage Dataset

***Figure 3.17** – Datasets from Images&PointClouds Cultural Heritage, source: (Pellis et al., 2025a)*

The Images&PointClouds Cultural Heritage Dataset (Pellis et al., 2025a) consists of five semantically labelled heritage point clouds from Florence, Italy, acquired through terrestrial photogrammetry at a density of 0.5 cm. Each point carries one of ten semantic labels covering the main architectural elements found in heritage buildings: arch, column, molding, floor, door/window, wall, stair, vault, roof, and other.

The dataset is shown in Figure 3.17 with label colors illustrated in Figure 3.18. Detailed statistics per class are given in Table 2. The five point clouds vary considerably in size and class distribution, with floors and walls consistently making up the largest share of points across all scenes, which is a relevant factor when interpreting the segmentation metrics.

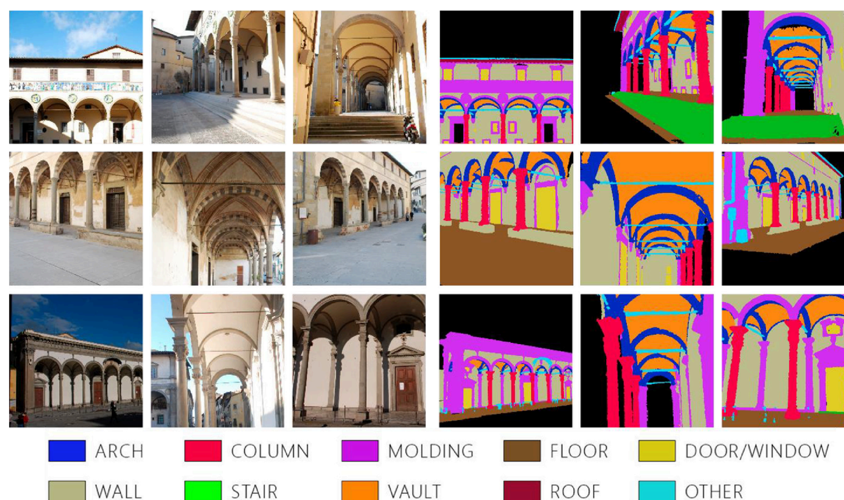


Figure 3.18 *– Images of the Images&PointClouds sites next to their colored labels, source: (Pellis et al., 2025a)*

Table 2 *– Statistics for the Images&Points Cultural Heritage Dataset*

Data	**Points**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
		Arch	Column	Molding	Floor	Door/ Window	Wall	Stair	Vault	Roof	Other
1_SC	16,316,690	6.7%	2.6%	15.7%	13.9%	5.2%	21.5%	3.9%	23.3%	5.1%	2.2%
2_OSA	7,123,548	6.8%	3.4%	4.9%	30.5%	8.4%	27.5%	0.0%	12.6%	3.4%	2.6%
3_SS	11,039,781	6.5%	6.2%	31.8%	23.8%	2.7%	16.4%	0.1%	11.1%	0.0%	1.4%
4_CG	19,306,811	0.6%	1.6%	2.8%	49.9%	0.2%	16.3%	0.0%	12.7%	11.5%	4.5%
5_CB	9,728,740	2.2%	7.1%	3.1%	18.7%	1.9%	24.3%	2.8%	13.4%	25.0%	1.7%
Total	63,515,570	4.1%	3.7%	11.4%	29.2%	3.1%	20.1%	1.4%	15.2%	9.0%	2.7%

4 Results

This chapter presents the results of the developed algorithms. The first part reviews the technical performance of the algorithms themselves by evaluating runtime, validity, and robustness. They will be testing during runs with varying inputs and parameters for both the erosion and the dilation. The second part concerns their effectiveness when applied to heritage point clouds. It assesses how well the algorithms perform during segmentation, object detection, and gap filling. These two parts together provide both a technical review of the algorithms and their applicability to real-world heritage use cases.

4.1 Algorithm review

The results are divided into three main criteria. First, the runtime reviews how fast the algorithm is and how processing time scales with input size. This gives an indication of the performance gains achieved when compared to the base algorithm. Afterwards, the validity is assessed to determine whether it produces the correct output. It verifies that the algorithm behaves as expected for a given input and structuring element. Finally, the robustness assesses how sensitive the output is to variation in input data and parameter choices. The different criteria both evaluate the standard algorithms as well as the density-aware and orientation-aware variants, both with and without erosion scores.

4.1.1 Erosion runtime

The first criterion considered is the processing time and how it scales with input size or other parameters. The total time indicates the overall performance achieved by each algorithm implementation. The time per point is used as a measure for the time complexity of each algorithm. Graphs generally use a log scale.

Runtime comparison with base algorithm

Figure 4.1 shows the processing time of the old sequential erosion algorithm and the new parallel implementation. The benchmark is run on a hollow cube of varying sizes, eroded with a 10×10 plane structuring element. At large input sizes, the new algorithm is around 100 times faster than the original. The difference grows as the input size increases, indicating a difference in time complexity between the two algorithms.

Looking at the time per point can tell us more about the exact time complexity. For the original algorithm, the time per point increases gradually with input size. This reflects the O(n log n) complexity of repeated kD-tree queries. As the point cloud grows, each nearest neighbor search takes longer because the tree is larger. For the new algorithm, the time per point initially decreases before stabilizing at a roughly constant value. The initial decrease is typical of GPU implementations. The parallel dispatch has a fixed overhead for uploading data, building the hash, and dispatching the compute shader. But as the input size increases, this overhead gets spread out over more points. Once the input is large enough, the overhead becomes negligible relative to the workload, and the time per point stabilizes, approaching O(n) time complexity.

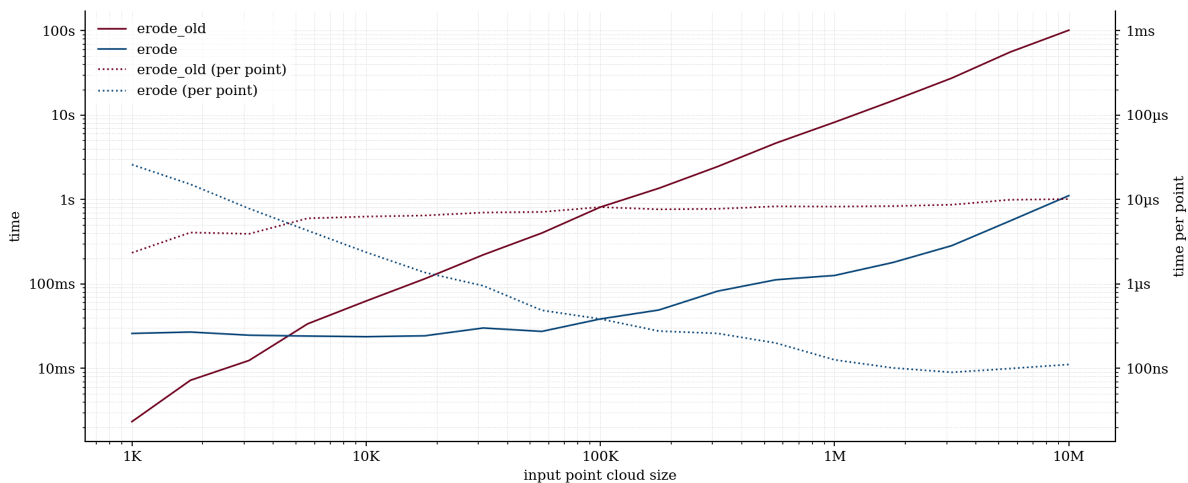


Figure 4.1 *– Total runtime and time per SE point for the base erosion algorithm compared to the new parallel implementation*

This is in line with the expectations of a spatial hash grid versus a kD-tree. Hash lookups have O(1) expected time, so processing n points in parallel results in linear time complexity, while kD-tree lookups have O(log n) expected time, resulting in log-linear complexity.

It should be noted that the exact performance figures are dependent on the hardware. The results described here were obtained on the hardware specified in Appendix A, but will vary across different GPU and CPU configurations. However, the scaling behavior should remain consistent, even if the exact runtimes and performance gains might differ.

Erosion variants benchmark

Figure 4.2 shows the total runtime and time per point for all erosion variants with increasing input sizes. All variants show roughly O(n) time complexity, consistent with the base erosion. Since each point is dispatched independently and the spatial hash lookup is approximately O(1) on average, the runtime scales linearly with the input size.

The density variant is the overall slowest algorithm. This is due to the additional density estimation check in the shader. Only a single density increment is used in this benchmark; the effects of increasing the density increments are explored later. Similarly, the brute-force variant

performs only a single rotation of the structuring element per invocation, but the fixed cost appears smaller. The score variants are consistently slower than their non-score counterparts. Since they compute the ratio of matches across all candidate neighbors without an early exit, the inner loop cannot break on the first mismatch. The biggest discrepancy is observed between the score and non-score variant of the brute-force erosion, as the initial coarse sweep has fewer points to check, making the early exit even more efficient. At the largest tested input size, the slowest variant runs approximately ten times slower than the base erosion.

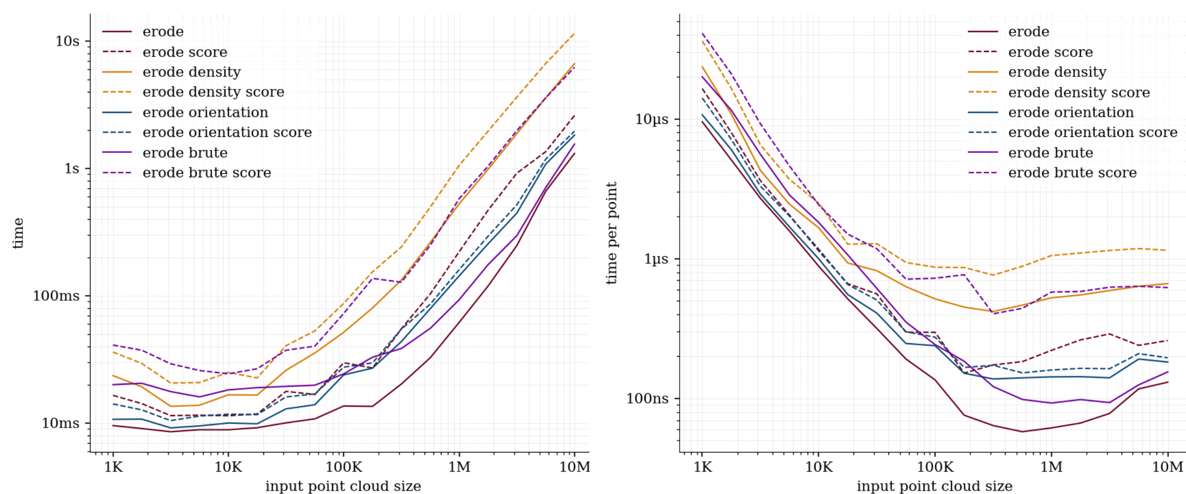


Figure 4.2 *– Total runtime and time per SE point for the erosion variants, total time on the left, time per point on the right*

In the time-per-point plot, the base erosion shows a slight upward inflection at the largest input sizes, indicating that it begins to saturate GPU throughput. The other variants reach this saturation point earlier in the input range, as their higher per-point costs mean they are operating near the device limit earlier. Since the overhead amortization effects have not fully ended yet, this uptick is less visible.

Structuring element benchmark

Figure 4.3 shows the runtime of the score erosion for structuring elements of increasing size, evaluated with fixed input data of a hollow cube of 10 million points. Two structuring element geometries are compared against each other: a plane with its point density increasing within a fixed spatial extent, and a line that grows by extending further in space. Both show a roughly linear increase in runtime with the number of structuring element points. This mirrors the O(n) behavior observed for an increase in input size, as each additional structuring element point introduces a fixed amount of work per input point in the form of an additional spatial hash lookup.

The line consistently outperforms the plane at equivalent sizes. These results show that the runtime of the erosion is not only dependent on the number of structuring element points, but also on their spatial distribution. A denser, spatially compact structuring element has a higher per-point cost than a sparser one of the same size.

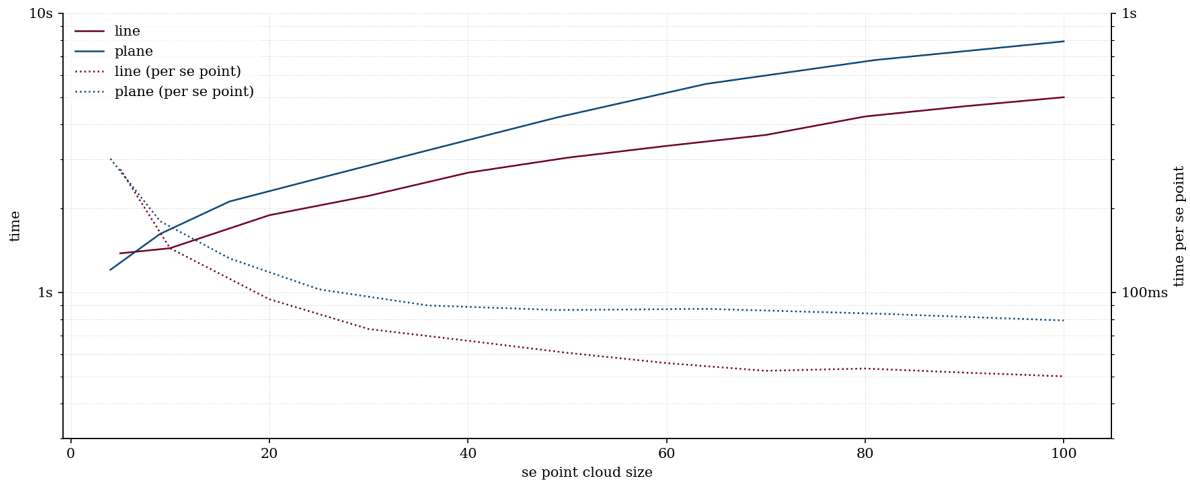


***Figure 4.3** – Total runtime and time per SE point for the score erosion with structuring elements of varying sizes and shapes*

Rotation benchmark

Figure 4.4 shows the total runtime and time per rotation for the brute-force orientation erosion and its score variant. It is evaluated on a 100K point hollow cube with a 10×10 plane structuring element. The pale lines show the approximate runtime of the regular erosion ran individually for the same number of rotations, serving as a reference for speedup obtained from the coarse sweep.

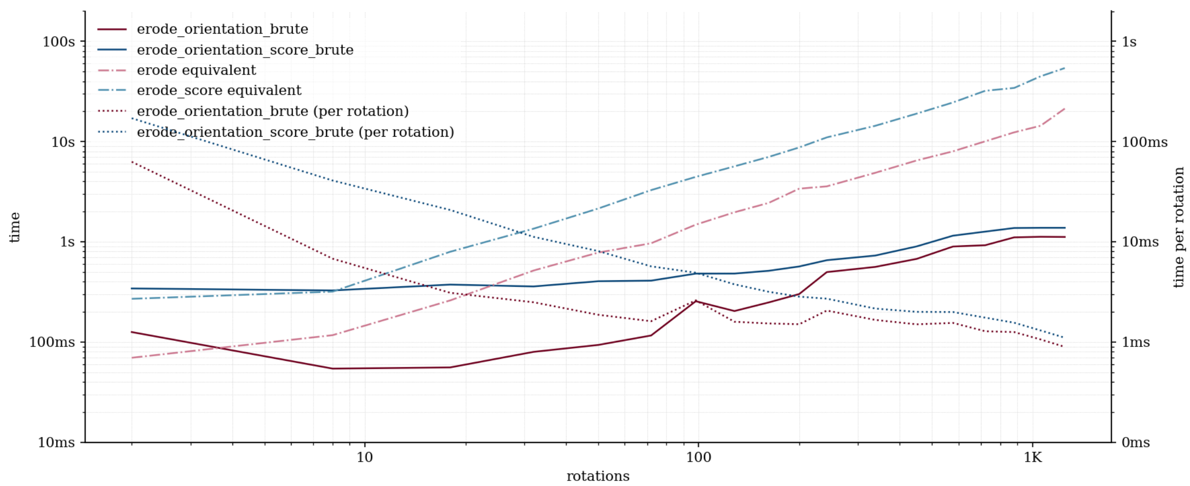


***Figure 4.4** – Total runtime and time per SE point for the brute-force orientation erosion for different numbers of sampled rotations. Pale dot-dash lines indicate the approximate time a regular erosion would require for the same rotation count*

Both variants show a roughly linear increase in total runtime with the number of rotations. The score variant runs consistently slower, starting at around 300ms versus 100ms for the regular brute-force. At low rotation counts, the per-rotation cost is high, as the GPU dispatch

overhead dominates. This decreases as more rotations are added, and the fixed cost is divided over more work. The two runtimes converge as rotation count increases, indicating that the relative overhead of the score variant diminishes as the rotation workload grows. At around 800 rotations, which is consistent with the theoretical cap of $(1.27\pi/0.1)^2 * 0.5 = 810$, the algorithm automatically limits the rotation count, causing the total runtime to plateau. After this point, additional requested rotations are discarded, and the runtime remains flat. The pale reference lines increase more steeply, which results in an approximately $10\times$ speedup for the coarse sweep implementation compared to the sequential erosions.

Density increment benchmark

Figure 4.5 shows the total runtime and time per structuring element point for the density-aware dilation as the combined structuring element size increases, as a result of adding more density increments. The input is a plane with a density gradient, and the structuring element consists of a 5×5 plane generated for spacings ranging from 0.1 to 1. The structuring element combines the increments of 0.1 symmetrically around the midpoint of the input density range, which is equal to 0.55.

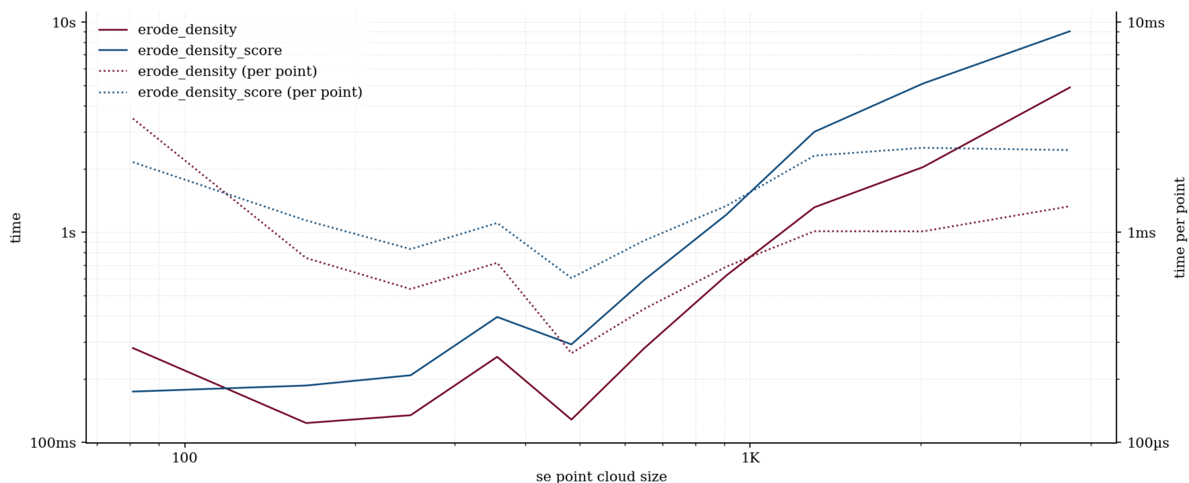


Figure 4.5 *– Total runtime and time per SE point for the density-aware dilation as a function of combined SE size, for an increasing number of density increments*

Initially, the runtime of the erosion algorithms decreases as the overhead costs get amortized across more structuring element points. At around 500 structuring element points, equivalent to 5 increments, the time per structuring element point starts to increase again, although stabilizing at larger sizes. This results in a longer processing time for the discrete structuring element implementation over the continuous one. The runtime could be reduced by limiting the number of checks required during a shader invocation. By storing how many points are in every density increment, the algorithm could automatically skip to the correct structuring element point indices, without having to compare the density of the structuring element point with the local density for every point.

4.1.2 Dilation runtime

Runtime comparison with base algorithm

Figure 4.6 shows the processing time of the original sequential dilation and the parallel GPU implementation, evaluated on a hollow cube of varying sizes dilated with a 5-point line structuring element. As the input size increases, the difference in processing time between the two algorithms widens. This indicates a difference in time complexity.

The time per point of the original algorithm increases linearly with input size, resulting in an overall quadratic $O(n^2)$ time complexity. As the input point cloud grows, each nearest-neighbor search takes longer because the KD-tree also becomes larger, and the newly inserted output points further slow down following searches as the groups get processed. Combined with the possibility of creating an imbalanced tree due to the naive insertion method, the runtime is severely impacted. At 100K points, the sequential algorithm already requires over 1000 seconds, making direct measurement at a million points impractical. The pale, dashed part of the graph, therefore, represents an extrapolation of the trend.

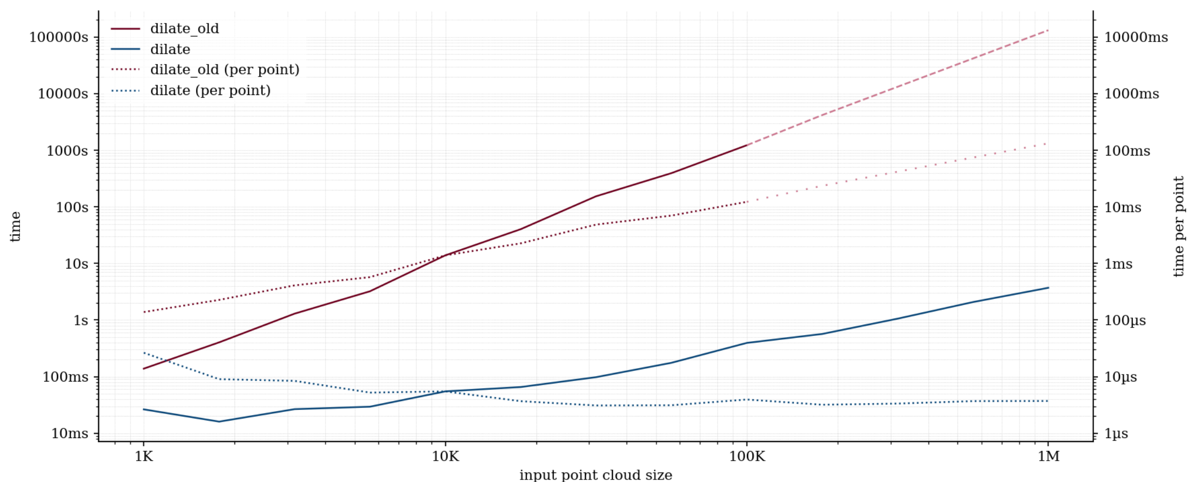


Figure 4.6 *– Total runtime and time per SE point comparison between the sequential and parallel dilation on a hollow cube with a 5-point line SE*

The time per point of the parallel implementation initially decreases before stabilizing at a roughly constant value, showing an $O(n)$ overall time complexity. This initial decrease is caused by the distribution of the GPU overhead over an increasing number of input points. Once the input is large enough, this overhead becomes negligible relative to the actual workload, and the time per point stabilizes. At 100K points, the parallel implementation is already more than 1000× faster, while at a million points, the estimated speedup exceeds 100,000×. The exact values are dependent on the setup, but the difference in time complexity will be present on all hardware configurations.

Dilation variants benchmark

Figure 4.7 shows the runtime of the three dilation variants: regular, density-aware, and orientation-aware, on a hollow cube with a 10×10 plane structuring element. All three show almost identical runtimes for varying input sizes. The small differences that do exist are smaller than those seen in the erosion variants.

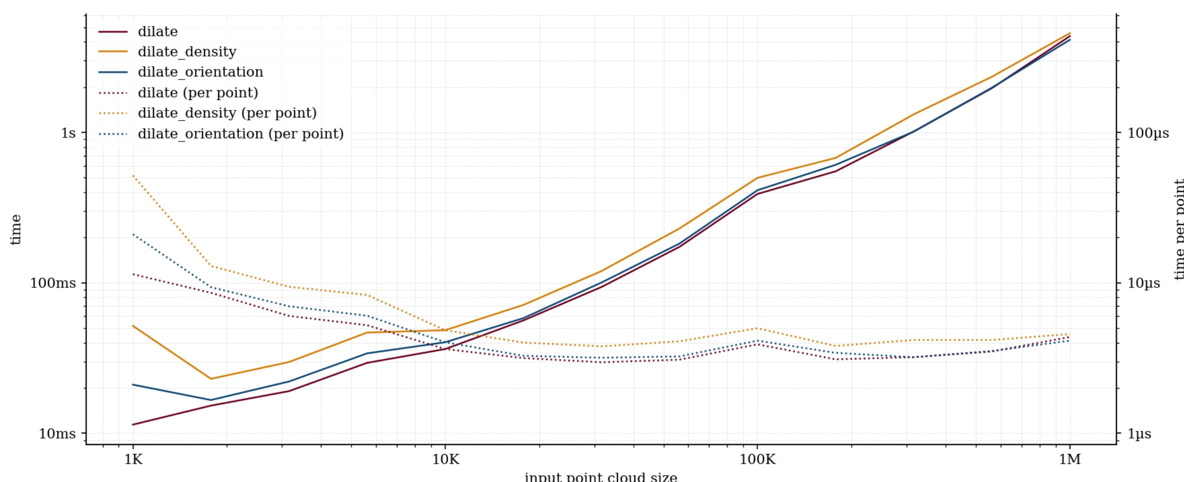


Figure 4.7 *– Total runtime and time per SE point of the dilation variants for increasing input size, evaluated on a hollow cube with a* 10×10 *plane SE*

The smaller difference can be explained by how the dilation dispatches work. Unlike the erosion, which dispatches one invocation per input point, the dilation dispatches one invocation per candidate output point. Since the number of candidates grows with both the input size and the structuring element size, the fixed overhead is spread more effectively from the start, reducing the time per point at smaller inputs. As a result, all three variants reach their stable time per point earlier and have runtimes close to each other for all input sizes. The density variant is slightly slower during this benchmark, as it has an additional density evaluation, but the difference is negligible when used in practice.

Structuring element benchmark

Figure 4.8 shows the total runtime and time per structuring element point for the dilation with a line and a plane structuring element of increasing size, evaluated on a fixed input of a hollow cube of 1 million points. The two geometries show a different time complexity, with the line more than 50× slower than the plane at 100 structuring element points.

The plane, of which the point density increases within a fixed spatial extent, shows almost constant total runtime across the varying structuring element sizes. It runs for around 2 seconds regardless of point count size. Its time per structuring element point decreases as expected, since the fixed grouping overhead is spread over more points within the same spatial area. Since the plane never grows larger in physical space, the set of potential conflicts in the grouping graph remains limited.

The line, on the other hand, has a very different runtime progression. Its total runtime grows linearly, from under 1 second at 5 structuring element points to around 80 seconds at 100, and its per-structuring element-point cost increases rather than decreases. As the line extends further in space with each additional point, it creates more conflicts in the input points, causing the grouping graph to grow in both size and connectivity. The sequential greedy coloring algorithm, used for grouping, becomes progressively more expensive as the number of edges between conflicting points increases.

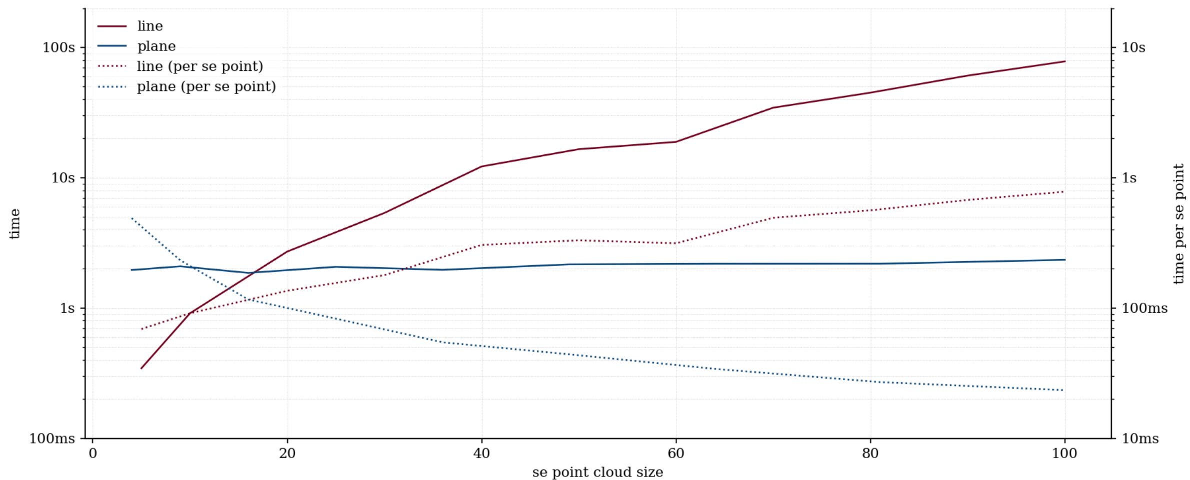


Figure 4.8 *– Total runtime and time per SE point for the dilation for a line and plane SE of increasing size, evaluated on a 1M point hollow cube*

This makes spatial extent an important factor for the dilation, in contrast to the erosion, where a spatially larger structuring element was actually somewhat beneficial due to more uniform hash access. A parallel graph coloring algorithm, such as the one introduced by Jones & Plassmann (1993), could reduce this issue for spatially large structuring elements, though the actual implementation falls outside the scope of this thesis.

Density increment benchmark

Figure 4.9 shows the total runtime and time per structuring element point for the density-aware dilation as the number of density increments increases, using the same structuring element configuration as the erosion density benchmark: a 5×5 plane generated for spacings from 0.1 to 1, combined symmetrically around the midpoint of the input density range.

The overall trend is similar to the erosion case. The total runtime increases slowly from around 8s to 15s while the time per structuring element point decreases, as the fixed dispatch overhead is spread over more structuring element points. However, the total runtime increase is slightly less pronounced than in the erosion, as every structuring element point must be loaded into the spatial hash and evaluated for every input point, regardless of whether it is active for a given density level. In the dilation, structuring element points that fall outside the local density range of a candidate are not inserted into the hash, reducing the memory size of the following passes and avoiding the upload cost. This keeps the marginal cost of adding further density

levels somewhat lower than in the erosion case, although the difference is quite small over the tested range.

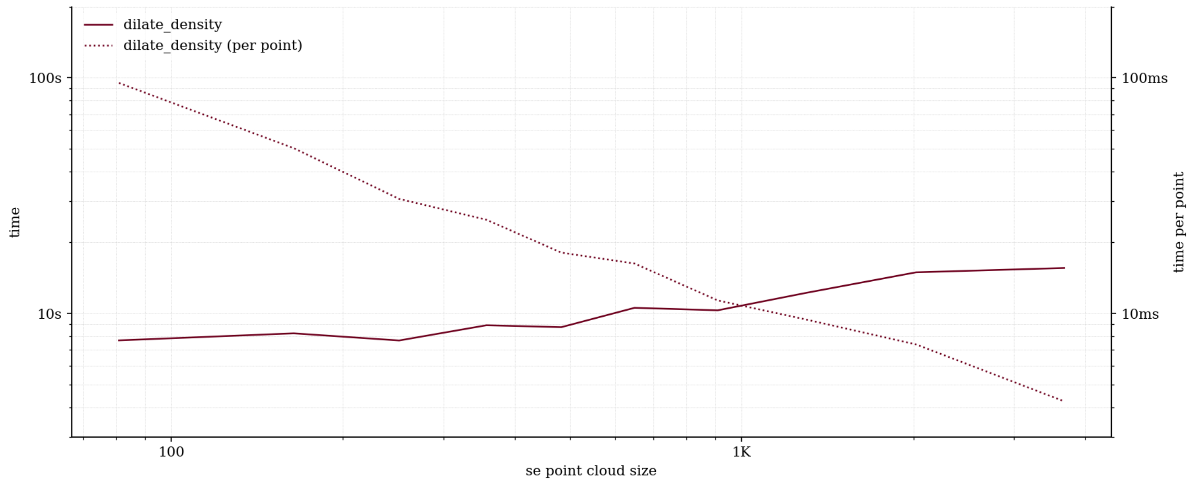


Figure 4.9 *– Total runtime and time per SE point for the density-aware dilation as a function of combined SE size, for an increasing number of density increments*

4.1.3 Erosion validity

The following tests verify that the erosion variants produce correct output for a set of controlled inputs. The goal is to make sure that the shader implementation is free of bugs and errors and that each variant behaves as expected before applying it to more complex scenarios.

Mathematical test

To verify the base erosion, a hollow cube with side length $n = 131$ is eroded with a 5×5 plane structuring element, with points arranged on a regular grid with a small random variance in the exact coordinate position. Because the geometry is mathematically regular, the expected point counts before and after erosion can be derived using simple formulas.

The number of points on the surface of a hollow cube with side length n is given by:

$$N = 6n^2 - 12n + 8 \tag{4.1}$$

For an erosion with a square structuring element with an odd side length m of which the origin is at the center point, only points that are at least $\left\lfloor \frac{m}{2} \right\rfloor$ steps away from every edge survive. The expected output size across the two exposed faces is:

$$N = 2(n - m - 1)^2 \tag{4.2}$$

Substituting $n = 131$ results in an input of 101,402 points and an expected erosion output of 32,258 points. As shown in Figure 4.10, both values match the algorithm output exactly, confirming that the structuring element is correctly applied and the shader logic is free of basic errors.

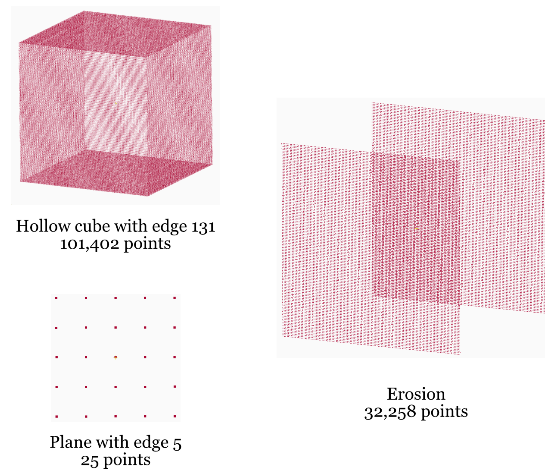


Figure 4.10 *– Mathematical validity test: hollow cube (101,402 points) eroded with a* 5×5 *plane SE, resulting in the expected 32,258 points*

Output comparison with base algorithm

Figure 4.11 compares the output of the original sequential erosion against the GPU-accelerated parallel implementation, both applied to the Stanford Armadillo model using a line structuring element of 5 points.

Both variants retain exactly 16,247 points and produce visually identical results. This confirms that the shader implementation does not introduce errors relative to the reference algorithm, and that the spatial hash correctly resolves point neighborhoods during parallel execution, even when applied to a more irregularly shaped point cloud.

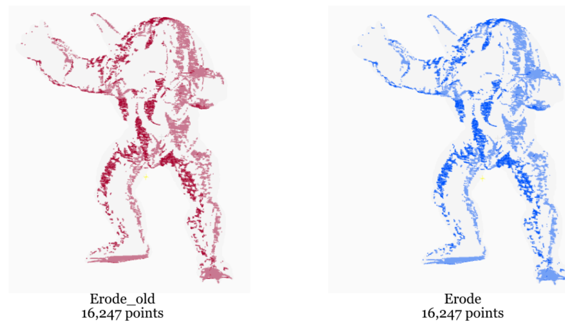


Figure 4.11 *– Erosion of the Stanford Armadillo with a 5-point line SE. The original sequential implementation (left) versus GPU-accelerated implementation (right), both resulting in 16,247 points*

Erosion score

Figure 4.12 shows the binary erosion and the score erosion applied to the Stanford Bunny with the same 5-point line structuring element. The number of points with a perfect score in the score output is identical to the number of points retained by the binary erosion, confirming that the two variants are consistent at the threshold boundary. Beyond that, the score output extends into regions that the binary erosion discards, assigning partial match values that decrease with distance from the fully matched region. This gives a more continuous indication of local

linearity in the direction of the SE than a binary result can provide. Because the SE consists of only 5 points, the score is discrete and limited to five distinct values.

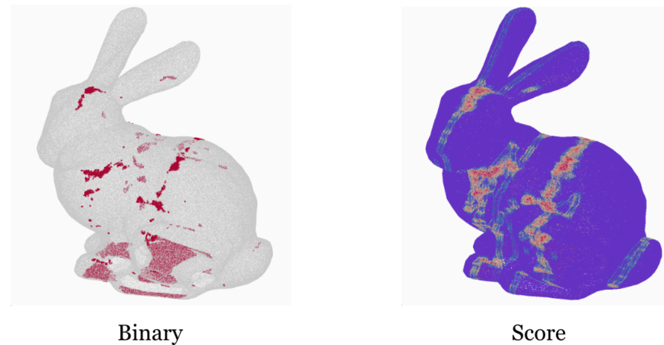


Figure 4.12 *– Binary vs. score erosion of the Stanford Bunny with a 5-point line SE. Score is mapped from blue (low) through red (perfect match)*

Density outcome

Figure 4.13 shows the result of eroding a plane of increasing density between 0.1 and 1 with a 50×50 plane with 0.1 spacing. This plane is downsampled at every 0.1 density increment for the discrete version. Both the discrete and continuous density erosion variants are evaluated.

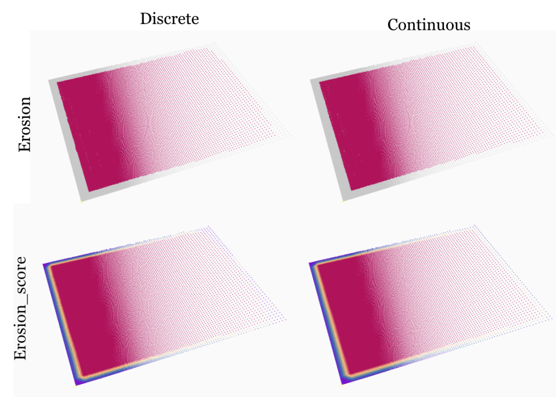


Figure 4.13 *– Density erosion on a density plane: discrete (left) and continuous (right) for both binary and score variants. Red indicates high match score, yellow medium, blue low*

The outputs are qualitatively similar, with both correctly identifying high-density and low-density regions with high scores, as should be expected. The continuous variant completes in approximately half the time of the discrete variant for this specific test, although the relative performance depends on the number of density increments used.

Orientation outcome

Figure 4.14 shows the results of the orientation-aware erosion variants applied to a hollow cube with a planar structuring element on the xz-plane. The base orientation erosion correctly retains points on all faces, which indicates the structuring element is correctly mapped to the local normal. The brute-force variant restricted to the z-axis correctly leaves out the top and bottom faces, since for those faces, no rotation about the z-axis can cause the structuring

element to align with a horizontal plane. Adding a tilt component to the rotation sweep does include these faces. This confirms that the orientation with a predefined direction and the rotation with an unknown direction both function as intended.

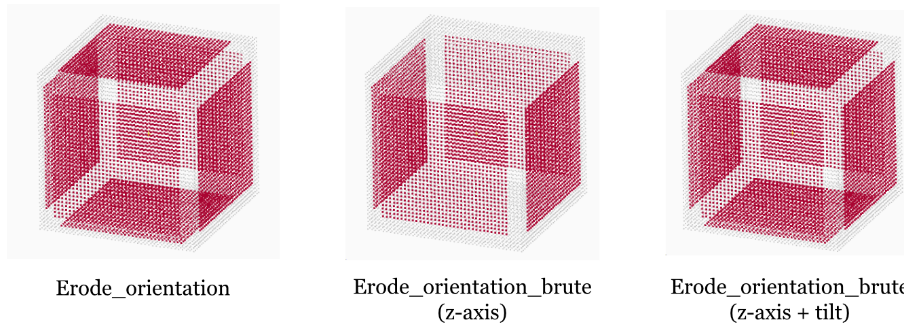


Figure 4.14 – *Orientation-aware erosion on a hollow cube: full orientation erosion (left), brute-force restricted to z-axis rotations (centre), brute-force with z-axis and tilt (right)*

4.1.4 Dilation validity

The validity of the dilation and its variants is verified through a set of controlled tests analogous to those applied to the erosion.

Grouping result

Figure 4.15 shows a close-up of the grouping output on a point cloud, with each group assigned a separate color. The groups are visually separated, and the minimum distance threshold between groups is respected when inspecting individual classes. In denser regions, new groups form at closer intervals, consistent with the behavior expected from the hash implementation. The size distribution of the groups follows a pattern typical of greedy algorithms. The first groups are of roughly comparable size, while the tail of the distribution contains a long sequence of very small groups, sometimes consisting of a single point. This is an inherent property of the greedy assignment strategy, which does not guarantee a balanced partition.

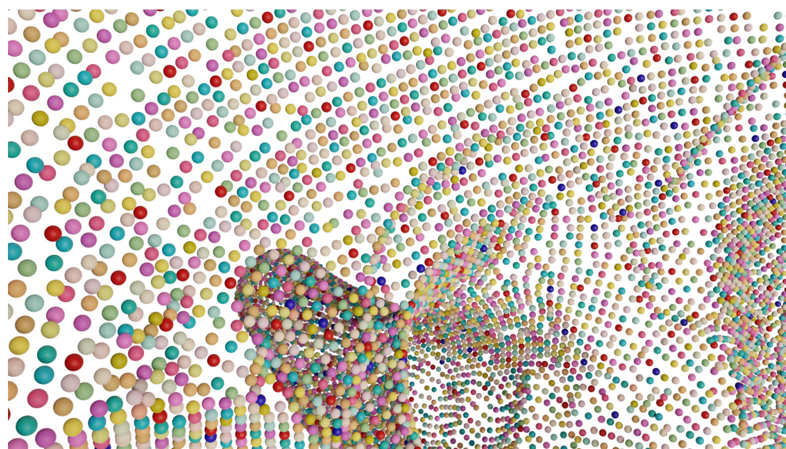


Figure 4.15 – *Close-up of grouping output; each color represents a distinct group*

Mathematical test

The dilation is verified using the same hollow cube with $n = 131$ and a 5×5 plane structuring element with $m = 5$. A small random variance is applied to the input points.

The input size is again given by:

$$N = 6n^2 - 12n + 8 \quad (4.3)$$

For the dilation, each face of the cube is expanded outward by the SE, and the corners and edges receive contributions from multiple faces. The expected output size is:

$$N = 2(n + m - 1)^2 + 4m(n - 1)(n - 2) \quad (4.4)$$

Substituting $n = 131$ and $m = 5$ gives an expected output of 371,850 points, which matches the algorithm output exactly, as shown in Figure 4.16. This confirms that the dilation correctly deals with race conditions, not generating duplicate points on geometrically simple input data.

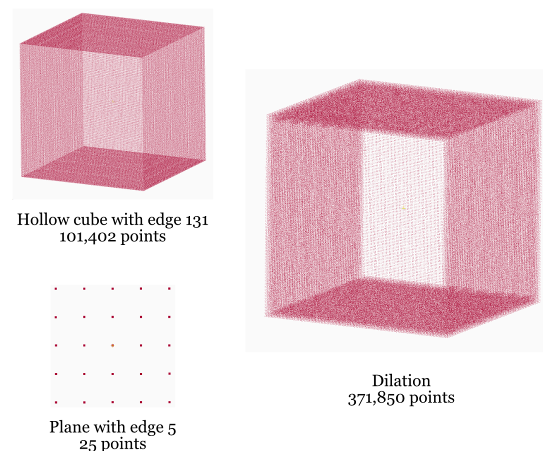


Figure 4.16 – *Mathematical validity test: hollow cube (101,402 points) dilated with a* 5×5 *plane SE, resulting in the expected 371,850 points*

Density test

When applying the dilation with a minimum distance threshold of 0.25, a small number of output points are found to be closer together than this threshold. As shown in Figure 4.17, approximately 1% of points fall below the threshold distance to their nearest neighbor. This violation does not indicate a fundamental error in the algorithm, but rather a known limitation of the hash-based insertion strategy. The distance check only considers the first point stored per grid cell. Points inserted into an already-occupied cell that overflows into a secondary bucket become invisible to this check. Since input points and structuring element points are inserted in parallel without mutual distance verification, any pre-existing spacing violations in the input or structuring element can continue and multiply in the output.

In practice, this issue does not arise when the input and structuring element are sampled such that all points occupy distinct grid cells, as demonstrated by the cube-plane test, but it becomes relevant for noisier or more densely sampled inputs. Two solutions are possible: subsampling

the input to approximately 1.5 times the minimum distance threshold before dilation, or applying post-processing to the output. Post-processing is generally preferable, as it can be applied selectively to only the newly inserted points in a closing operation, where the number of affected points is typically small. A similar post-processing step was used in the original reference algorithm.

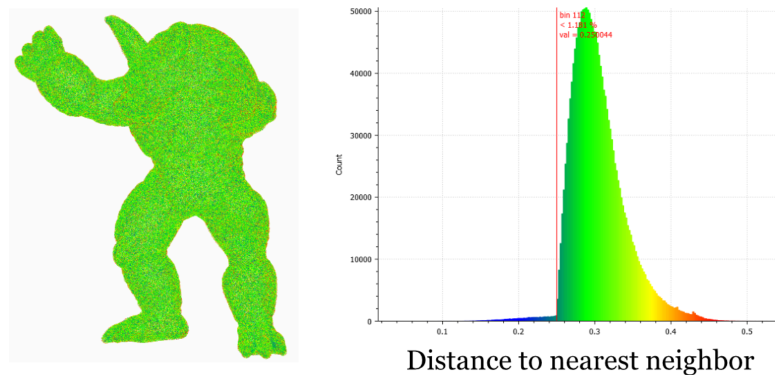


Figure 4.17 *– Nearest-neighbour distance histogram for the dilation output. The red line marks the 0.25 threshold; 1.18% of points fall below it*

In the context of heritage applications, this issue is only relevant for gap filling, which uses the closing operation and retains a subset of the dilated points. Operations such as the opening or the hit-or-miss transform return to the original point set after the morphological operation, so any spacing violations in intermediate results are ignored.

Output comparison with base algorithm

Figure 4.18 shows the difference between the output of the original sequential dilation of a solid sphere with a smaller one, and the parallel implementation on the GPU. The two outputs are not identical because the grouping step processes points in a different order depending on parallelism and hash insertion sequence.

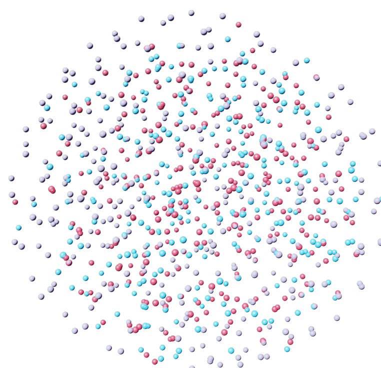


Figure 4.18 *– Difference between the original (red) and the parallel (blue) dilation outputs, with shared points colored purple*

The set of points selected per group can differ slightly between runs. The total point counts are also marginally different between the two implementations, though always within a comparable range. This is an expected consequence of the non-deterministic ordering inherent to the parallel hash and does not reflect a correctness error in either implementation.

Density outcomes

Figure 4.19 shows the result of dilating a density plane with a 50×50 plane SE, for both the discrete and continuous density variants. Both correctly fill in regions consistent with the local density of the plane, with the color gradient progressing from low-density (red) to high-density (purple) areas as expected. The discrete variant produces a somewhat cleaner result at the borders, where relatively little infill from overlapping structuring elements occurs. In the interior, and particularly in the context of a closing operation where these border points would subsequently be eroded away, the difference between the two variants is less relevant. Both variants accurately reflect the density of the underlying plane.

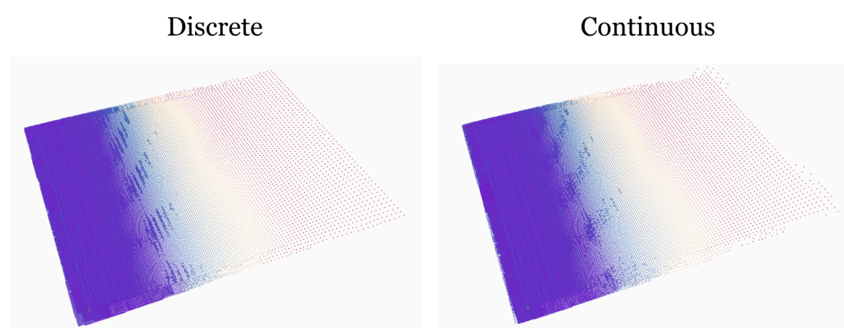


Figure 4.19 *– Density-aware dilation on a density plane: discrete (left) and continuous (right)*

Orientation outcomes

Figure 4.20 shows the orientation-aware dilation and closing applied to a hollow cube with holes on all faces. The input normals are correctly estimated even on incomplete faces, allowing the dilation to propagate points outward in the direction consistent with each face's orientation. The orientation-aware closing then applies the subsequent erosion using the transferred normals, correctly restoring the cube's surface geometry. This confirms that the normal estimation and transfer between the dilation and erosion steps remain stable even when the input geometry is incomplete.

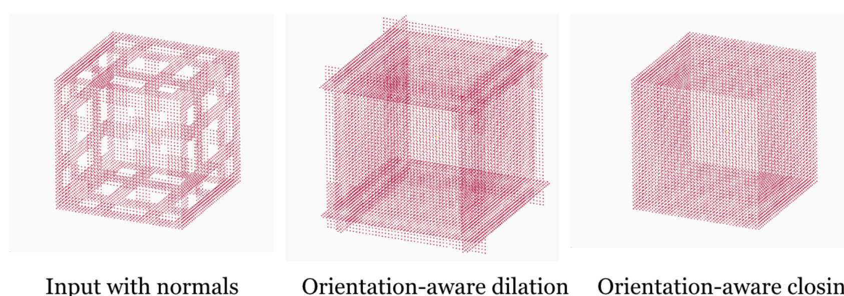


Figure 4.20 *– Orientation-aware dilation and closing on a hollow cube with holes: input with estimated normals (left), dilation result (center), closing result (right)*

4.1.5 Erosion robustness

The following tests check how the erosion algorithm behaves under extreme or unusual inputs and parameter choices.

Input size

The erosion has been verified to handle the erosion of a point cloud of 30 million points with a structuring element of 500 points, without any major issues. Memory usage is determined primarily by the two buffers that must be stored on the GPU at the same time, namely the spatial hash of the input point cloud and the output buffer. Because the erosion does not generate new points, the output is always equal to or smaller than the input. Very large structuring elements need more chunking of the input data. At 10,000 structuring element points, for example, the chunk size is reduced to 500, increasing the number of passes over the input, but it does keep the memory within limits for every pass.

Threshold

Figure 4.21 shows the effect of varying the distance threshold on the score erosion of the Stanford Bunny. At a very small threshold of 0.01, the hash cells are very fine relative to the average spacing, and the algorithm detects very few points; in the best case, there are some whispers of detection.

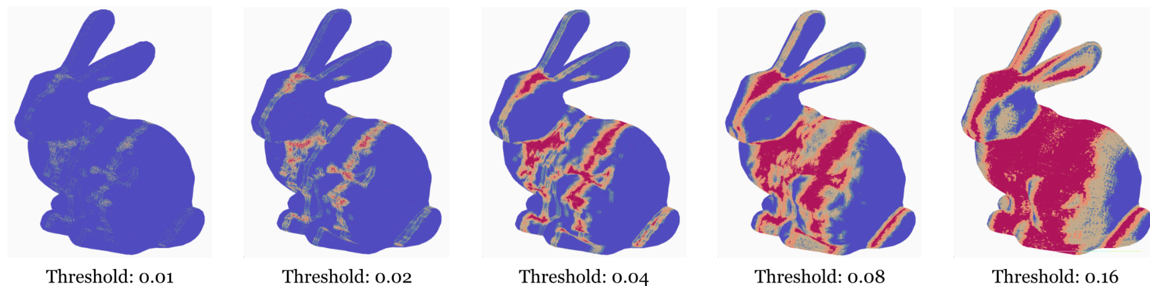


Figure 4.21 *– Score erosion of the Stanford Bunny with a 5-point line SE at increasing distance thresholds. Blue indicates a low score, red a high score*

As the threshold increases, more of the surface is matched. At 0.16, large continuous regions receive high scores regardless of local geometry, and the load factor of the hash table increases, limiting performance. A threshold set slightly smaller than the average point spacing generally results in the most meaningful results, as thresholds larger than the average spacing tend to over-detect. Smaller thresholds have a small increase in runtime as well due to finer cells, but this does not correspond to an improvement in detection quality.

Orientations

The number of rotations checked in the brute-force orientation erosion is bounded by the estimated circumference of the structuring element divided by the minimum distance threshold, with an absolute minimum of one degree per step. Checking more angles than the circumference limit does not provide any detection gains, as the resulting angle increments are smaller than the spatial precision of the hash. When working with very large structuring elements, using

the full 360×180 possible orientations can lead to degraded performance or a GPU crash, as the per-point workload becomes too high.

4.1.6 Dilation robustness

Input size
The dilation has been verified on a point cloud of 30 million points with a 32-point SE. However, the output is capped at 50 million points, meaning any points added to the hash beyond this point are discarded. The primary limiting factor is GPU memory, as the spatial hash must be fully stored on the GPU when running the shader. The available VRAM determines the maximum hash size and, therefore, the maximum input that can be processed in a single pass. Further chunking of the input in a way that only parts of the input are uploaded to the hash is technically possible, but requires stitching the output chunks back together, which is outside the current scope.

Threshold
The dilation threshold sets the minimum distance between inserted output points. At small thresholds, nearly all candidate points are inserted since the distance check is easily satisfied. At large thresholds, especially those way larger than the average spacing of the input, the algorithm may fail to insert any new points at all, or produce geometric artefacts, since it assumes that points already in the hash and the points within a structuring element are spaced at least as far apart to satisfy the threshold. The hash visibility issue described in the density test is exacerbated at high thresholds since only the first occupant per cell is visible to the distance check. A high threshold increases the chance that an invalid insertion candidate is still allowed, since it cannot be verified against the invisible neighbor. Threshold choice has a limited impact on runtime; a slight speedup occurs at smaller thresholds due to fewer successful insertions, but all candidates are still evaluated regardless.

The density-aware variant calculates the threshold automatically from local density estimates, which makes it dependent on the local density range of the input. Areas that are substantially denser than the rest, such as regions containing duplicate points, can produce unexpectedly small threshold values that affect the surrounding cell layout. It is therefore recommended to remove duplicate points to a minimum spacing before applying the density dilation.

Overall, the best performance is achieved at a threshold slightly smaller than the average spacing of the point cloud, with a structuring element that is no denser than the input point cloud.

4.2 Heritage use case

This section demonstrates that the implementation developed in this thesis resolves the problems of mathematical morphology when applied to heritage point clouds as defined earlier in this thesis. This is done by applying the morphological operations to real-world heritage point clouds across three use cases: segmentation, object detection, and gap filling. The tests are kept simple and should be treated as proof-of-concept implementations instead of fully developed workflows. The goal is not to present an optimized pipeline but to show that mathematical morphology, as implemented in this thesis, can be applied and is useful for practical heritage processing tasks.

4.2.1 Segmentation

The segmentation workflow involves applying several openings or hit-or-miss transforms sequentially, and removing matched points from the input before each following step. This allows the different categories to be segmented using specific structuring elements, with orientation-aware and brute-force variants used where objects might be rotated in space.

Table 3 *– Segmentation metrics for the Images&PointClouds dataset*

Data	mIoU	OA	mPrec	mRec	mF1
1_SC	0.491	0.675	0.663	0.609	0.599
2_OSA	0.714	0.870	0.807	0.837	0.818
3_SS	0.656	0.816	0.757	0.806	0.767
4_CG	0.711	0.904	0.825	0.809	0.812
5_CB	0.591	0.803	0.698	0.720	0.683
Average	0.632	0.813	0.750	0.756	0.736

Table 3 shows the segmentation metrics obtained on the different point clouds from the Images&PointClouds dataset. An average mIoU of 0.632 was achieved, which proves that a moderately good segmentation with a simple implementation using morphology can be obtained. The overall accuracy of 0.813 indicates that approximately 81% of points are correctly classified. However, the gap between overall accuracy and the mean precision and recall shows that there is class imbalance present. The large classes, such as the floor, roof, and stairs, are classified well, while smaller and harder to separate classes, such as moldings, drag down the average class scores. The F1 score is also higher than the mean precision and recall, which confirms that the best performance is obtained in larger classes. Overall, the results are comparable to or slightly better than the segmentation obtained from the classification of images and subsequent reprojection presented in the original paper (Pellis et al., 2025b), although the random forest approach of (Grilli et al., 2019) obtains better results using geometry-based input features, albeit on a different dataset.

The different datasets also return varying results. Some point clouds feature more noise, making accurate detection harder. Furthermore, the complexity and uniformness of every label

also plays a role. For example, 1_SC features many unique windows and moldings that are hard to capture with limited structuring elements, while 4_CG features more repetitive geometries, easily captured in a single structuring element. Finally, the class balance also plays a role. Since floors are easy to detect, the 4_CG dataset — comprised of 50% floors — has an easier time obtaining a high overall accuracy, while a dataset like 1_SC, having proportionally more difficult classes, such as moldings, struggles to obtain a high score.

The visual result in Figure 4.22 shows several limitations of the approach. Large portions of the roof are not detected, which can mainly be attributed to the noise and the artificial surface reconstruction present in this dataset, rather than a real failure of the morphological operations themselves. The sequential implementation of the workflow introduces a further issue. Points classified, and therefore deleted, by an earlier segmentation step are permanently removed from the input of later steps. For classes such as walls and moldings, which share geometric overlap and are often processed later in the sequence, this deletion causes some regions to go undetected even though an opening applied to the full point cloud would have matched them correctly.

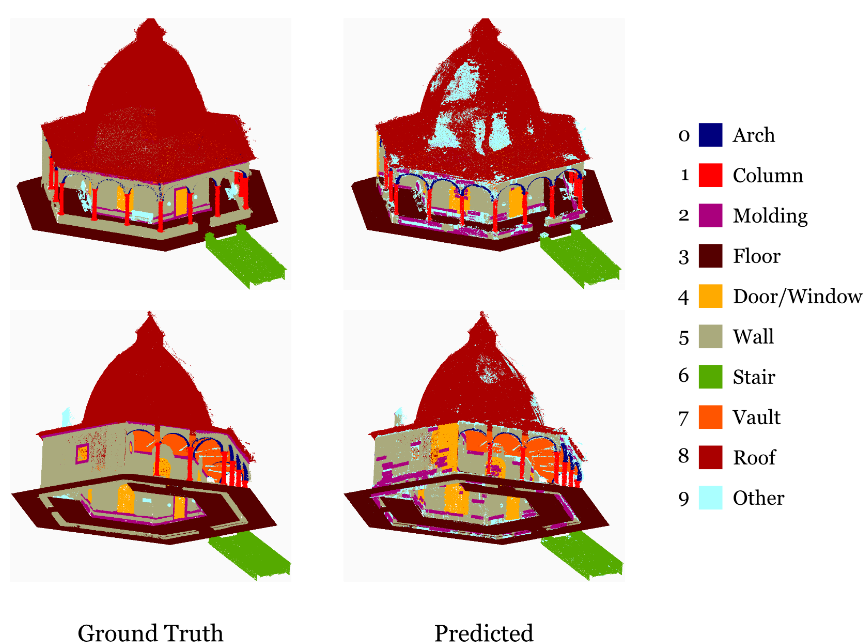


Figure 4.22 *– Segmentation results for 5_CB: ground truth (left) and predicted (right)*

Another issue with the current approach is the presence of a large number of unclassified points. A scoring mechanism that assigns every point a likelihood of belonging to a specific class would reduce the amount of leftover points and make the segmentation more robust. However, obtaining such a score from morphological operations directly is not straightforward, but not impossible. When combining these scores with more generic geometric features such as surface variance or sphericity, these scores could serve as input to a random forest classifier as used in the paper by Grilli et al. (2019). Combining mathematical morphology with machine learning classification has the potential to significantly improve segmentation.

Figure 4.23 illustrates a third issue that affects object-level detection using the opening or hit-or-miss transform. The structuring element used for pillar detection produces high scores not only at points that match the pillar structuring element origin but also at surrounding points, since partial matches are rewarded by the score erosion. To ensure all pillars are detected, the score threshold must be set low enough to include these partial detection points, but this causes the following dilation to overshoot the true pillar extent. As visible in the figure, some detected pillars are taller than their ground truth counterparts as a result. This presents a trade-off between a stricter score threshold, which reduces overshoot but risks missing objects entirely.

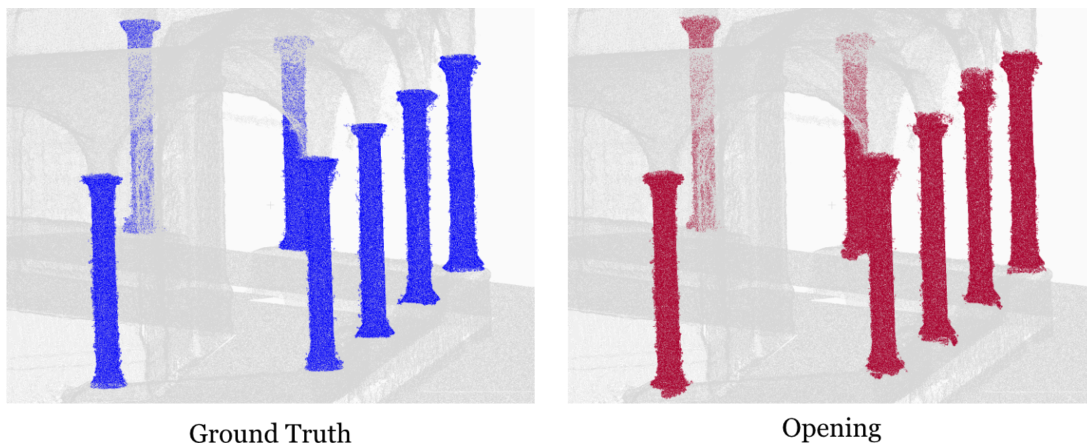


Figure 4.23 *– Pillar segmentation obtained with the opening for 5_CB: ground truth (left) and opening result (right)*

One solution would be to post-filter the score output to retain only local maxima before applying the dilation. This would guarantee one erosion seed point per detected object. This approach is most effective when the structuring element represents the full object geometry rather than a geometric approximation such as a line or circle. However, implementing such a post-filter would require an additional processing step outside the current scope.

Overall, the results demonstrate that a straightforward implementation of mathematical morphology is capable of meaningful semantic segmentation of heritage point clouds. The performance already approaches the results observed in other papers. With more refined structuring elements, better score-based filtering, and integration into a more complete classification workflow, the approach has clear potential for practical segmentation uses.

4.2.2 Object detection

The object detection tests apply the opening and hit-or-miss transform on two datasets, the Fontana dei Mesi (FM) and Houghton Hall (HH). They use structuring elements mainly obtained from sampled objects. Each test demonstrates a different variant of the morphological operations, targeting objects that require different handling of orientation, density, and negative space. The results are summarized in Table 4.

Regular hit-or-miss

Figure 4.24 shows the detection of pillars in the FM dataset using the hit-or-miss transform. The positive structuring element is a sampled pillar, and the negative structuring element represents the hollow interior. This limits matches to where the space inside the pillar is empty, mainly to avoid false positives in foliage-dense areas. The method achieves an F1 of 0.928, detecting pillars reliably along the left side of the fountain. The opposite side of the fountain was not scanned from the back and is therefore geometrically incomplete. The undetected pillars on that side are not counted as false negatives.

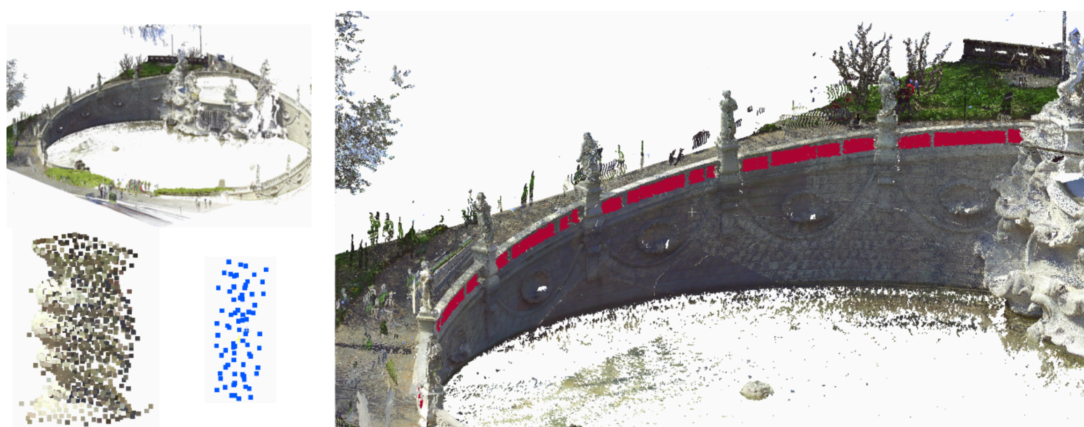


Figure 4.24 *– Pillar detection in the FM dataset using the hit-or-miss transform. Red indicates detected pillars*

An important result is that the hit-or-miss successfully avoids most foliage, which a regular opening would have matched. The square posts between the round pillars also go undetected, demonstrating that the specificity of the hit-or-miss is sufficient to discriminate between geometrically similar but still distinct objects. The small number of false positives and negatives can partly be attributed to the noise in the structuring element itself, which was sampled directly from the point cloud.

Brute-force opening

Figure 4.25 shows the detection of decorative shells in the FM dataset using the brute-force orientation opening. The curved surface of the shells produces unreliable local normals, making the orientation-aware variant based on predetermined normals unsuitable. The brute-force variant, sweeping at 15° increments around the z-axis, avoids this issue and successfully detects all 12 shells. It achieves a perfect recall of 1.000 and an F1 of 0.923. A coarse increment is sufficient as the shells do not require very small rotation increments to produce a satisfactory match. The two false positives that are present appear in the foliage. Without a negative structuring element to limit the match to specific geometry using negative space, the highly irregular nature of vegetation can, by pure chance, find a match for a shell-shaped structuring element during the opening. Despite this, the low amount of false positives, even with the number of rotations checked, is a good sign, since each additional rotation increment creates a new opportunity for these kinds of false matches to be found.

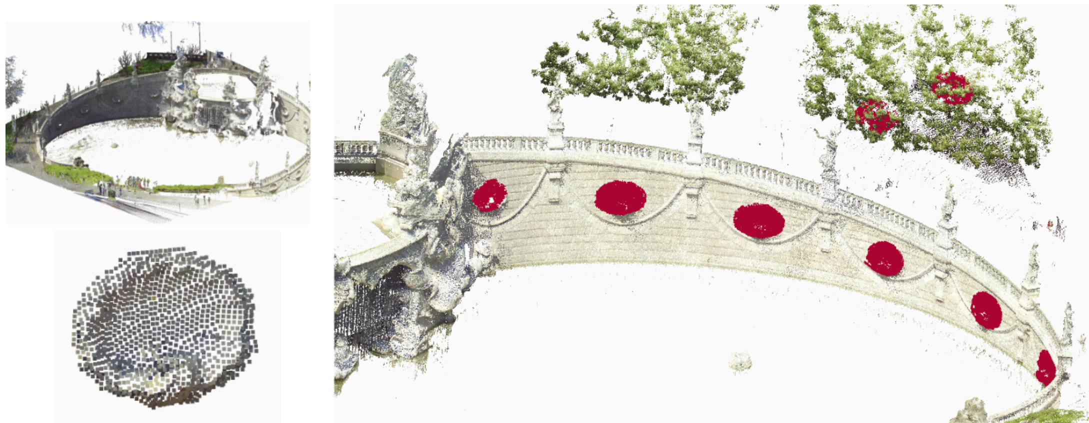


***Figure 4.25** – Shell detection in the FM dataset using the brute-force orientation opening. Red indicates detections.*

Orientation-aware hit-or-miss

Figure 4.26 shows the detection of windows in the HH dataset using the orientation-aware hit-or-miss transform. The positive structuring element is a sampled window frame, and the negative structuring element is constructed by dilating the window outline with a line perpendicular to its face, and then subtracting the original points, effectively making it so the glass parts of the window have to be empty. The method performs better than the original algorithm as tested in Section 2.4. The original test also needed a smaller structuring element and produced more false positives. The new algorithm achieves an F1 score of 0.668, which is a significant improvement, but also worse than the FM dataset. Several windows on the left facade contain points in the glass parts, which prevent a match even when the window frame is correctly detected by the structuring element. Especially rougher wall sections, including the brickwork at the base and the roof molding at the top, produce the most false positives as the chance of randomly satisfying the hit-or-miss is higher here.

***Figure 4.26** – Window detection in the HH dataset using the orientation-aware hit-or-miss transform*

Density-aware opening
Figure 4.27 shows the detection of decorative acorns on the HH roof using the density-aware opening, comparing the discrete and continuous structuring element variants. The discrete variant achieves an F1 of 0.720 with a high precision of 0.900, but limited recall of 0.600, while the continuous variant achieves an F1 of only 0.442, with a higher false positive count of 30. The continuous structuring element is less strict in what it matches and triggers on any acorn-like geometry in the scene, regardless of density. This generates a large number of false detections in low-density regions. The discrete variant is considerably more selective, but struggles in areas where the local point density is too low to satisfy the structuring element requirements, particularly for acorns near the edges of scan coverage.

A structural limitation affects both variants: the origin of the structuring element is placed at the topmost point of the acorn, which is the most geometrically central but also the least reliably captured in the scan. If this point is absent due to occlusion or scan angle, the object goes undetected regardless of how well the remainder of the structuring element matches. The two variants also detect partially different sets of acorns, highlighting how sensitive object detection is to the exact construction of the structuring element and the placement of its origin.

Figure 4.27 *– Acorn detection on the HH facade using the density-aware opening: discrete (top) and continuous (bottom)*

Overall statistics
Table 4 summarizes the object detection results across both datasets. Performance in the FM dataset is consistently higher, with F1 scores above 0.920 for both pillars and shells, reflecting the higher point density and relatively clean geometry of that dataset. Results for HH are generally lower, partly because the test cases are inherently more demanding as windows require precise negative space constraints and acorns are small and sensitive to noise, partly because the HH point cloud has more noise, especially around the window frames.

Across all tests, the discrete structuring element produces more consistent and precise results than the continuous variant, although it had a lower recall in sparse regions. The score filtering used does introduce a tradeoff. A lower filter score increases recall at the expense of precision. Although the F1 score does account for this, it is not confirmed that the optimal F1 score is

obtained from a specific score filter, unless you check the entire possible range. The placement of the structuring element origin also has an impact on recall, particularly for small objects, where a single missing point can cause a non-detection. Despite these limitations, the results demonstrate that morphological object detection on real-world heritage point clouds is viable and can achieve strong performance when the structuring element is carefully created and the appropriate variant is selected for the orientation and density requirements of the target object.

Table 4 *– Object detection metrics for the FM and HH datasets with different objects*

Data	SE	TP	FP	FN	Prec.	Recall	F1
FM	Pillar	109	7	10	0.940	0.916	0.928
FM	Shell	12	2	0	0.857	1.000	0.923
HH	Window	104	38	65	0.732	0.615	0.668
HH	Acorn Disc.	18	2	12	0.900	0.600	0.720
HH	Acorn Cont.	17	30	13	0.362	0.567	0.442

4.2.3 Gap filling

Gap filling is evaluated using three different tasks: surface reconstruction using the closing operation, edge detection using the orientation-aware hit-or-miss transform, and object reconstruction using template matching. Together, these could form the basic structure of a workflow in which holes are first found, and then filled either using a more global approach using the closing or a more object-specific one using template matching. Here, these three tasks are evaluated separately.

Surface reconstruction

For surface reconstruction, the closing is applied to the PB dataset using a range of line and plane structuring elements, with point spacing set to approximately double the average input spacing. This discrepancy allows for a spatially larger structuring element to be used, as overlapping dilated regions will naturally fill the gaps left by individual structuring elements. As such, the minimum distance threshold is kept at 0.5 for both the dilation and erosion steps. Points that land close to but not exactly on the original surface are counted as interpolated. They densify the cloud and are classified as neither correct nor incorrect.

Table 5 shows that as structuring element size increases, recall rises across all shapes, but precision drops sharply, lowering the overall F1 score. This highlights the trade-off at play. Larger structuring elements are able to bridge wider gaps but become increasingly indiscriminate, filling recessed areas such as windows that are not holes at all. The chamfer distance remains relatively stable throughout, indicating that even the false positives are geometrically close to the surface and that no extreme spatial errors are introduced.

Table 5 *– Results of closing with different SEs for gap-filling in the PB point cloud*

SE	Inter.	TP	FP	FN	CD	Prec.	Recall	F1
line x 9	375794	67802	249781	174922	0.821	0.213	0.279	0.242
line x 15	487867	79106	391988	163618	0.816	0.168	0.326	0.222
line x 21	556013	88359	547769	154365	0.810	0.139	0.364	0.201
line x 27	604611	94685	713219	148039	0.806	0.117	0.390	0.180
line z 9	356503	67244	219248	175480	0.844	0.235	0.277	0.254
line z 15	479470	75906	326257	166818	0.838	0.189	0.313	0.235
line z 21	550733	81863	440541	160861	0.830	0.157	0.337	0.214
line z 27	596525	83832	552860	158892	0.822	0.132	0.345	0.191
plane 5	538165	79650	258986	163074	0.818	0.235	0.328	0.274
plane 9	700033	99750	548075	142974	0.795	0.154	0.411	0.224
plane 13	737234	111622	918590	131102	0.772	0.108	0.460	0.175
plane 17	696197	126108	1383356	116616	0.753	0.084	0.520	0.144

Comparing structuring element shapes, lines oriented along the z-axis obtain better results than those along the x-axis at the same sizes. This can be attributed to the geometry of the building itself. Recesses in the x-direction — corresponding to features roughly parallel to the floor — are more prevalent and are more easily mistaken for holes by an x-oriented structuring element, which introduces more false positives. Small plane structuring elements outperform lines of similar sizes, as they achieve a higher recall with similar precision. This suggests that two-dimensional infill is better at bridging gaps across a surface. However, larger planes quickly start overfilling. A 17×17 plane has a recall of 0.520 but a precision of only 0.084, indicating that at that scale the closing fills far more recesses than actual holes.

As visible in Figure 4.28, the 15-point x-direction line handles small localized gaps reasonably well, but large gaps stay untouched. Curved surfaces, such as the dome, are particularly problematic as points inserted by the dilation are more likely to land slightly above or below the true surface, increasing the interpolated point count and reducing precision. Corners present a further challenge, since correctly filling them would require direction-specific structuring elements for each axis simultaneously, substantially increasing both runtime and the number of inserted points. The orientation-aware closing handles the column sections more cleanly than the regular closing, avoiding the wrong infill between columns that the regular dilation produces. This is explained by the fact that the orientation-aware dilation correctly restricts point insertion to directions consistent with the local surface orientation.

In conclusion, surface reconstruction via closing can patch small, smooth gaps but faces an inherent trade-off between the size of gaps it can fill and the tendency to incorrectly fill recessed areas. The approach performs least well in areas with complex and intricate geometry, where

gap filling is most critical. This suggests that a more local strategy, such as first detecting holes and then filling them selectively, would be more effective.

Figure 4.28 *– Gap filling visual results for the PB dataset using a 15-point line SE in the x-direction. Red indicates inserted points*

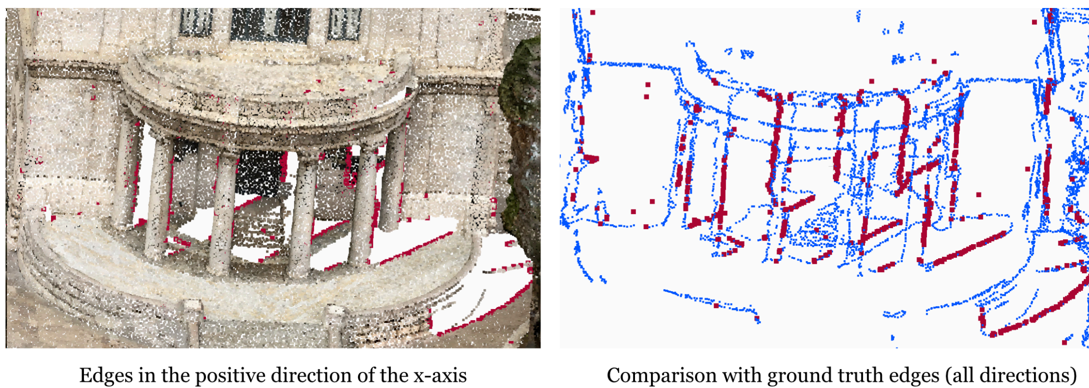
Edge detection
Edge detection is performed using an orientation-aware hit-or-miss transform with directional structuring elements targeting surface boundaries in four directions: positive and negative x and z. Each structuring element is designed to match points that lie on one side of a surface boundary without a corresponding neighbor on the other side, effectively identifying scan occlusion edges.

Table 6 *– Results of orientation-aware erosion with different SEs for edge detection in the PB point cloud*

Direction	TP	FP	FN	Precision	Recall	F1
X +	5723	2433	52799	0.702	0.098	0.172
X -	5418	2422	53104	0.691	0.093	0.163
Z +	4115	1461	54407	0.738	0.070	0.128
Z -	4526	1671	53996	0.730	0.077	0.140
All	16593	7612	41929	0.686	0.284	0.401

Table 6 shows that individual directional structuring elements achieve high precision, consistently above 0.69, but very low recall, as each SE only detects edges oriented in its specific direction. This is expected: a boundary running perpendicular to the x-axis will not be detected by a z-axis SE. When the detections from all four directions are combined, recall rises substantially to 0.284, confirming that the four structuring elements are targeting different subsets of the edge population. The combined precision takes a modest hit to 0.686 as the false positives from each direction accumulate, but the overall F1 of 0.401 is considerably better than any individual direction.

Figure 4.29 illustrates that the false positives are not randomly distributed. They cluster close to the true edge locations, with few extreme spatial outliers. This is visible in the ground truth comparison, where the detected edges (blue) and ground truth (red) overlap along the main structural boundaries of the column area. The remaining false positives largely correspond to surface discontinuities that are geometrically similar to edges but not labelled as such in the ground truth, such as transitions between decorative elements. This suggests that with more finely tuned structuring elements and post-processing to eliminate near-duplicate detections, the precision could be improved further without sacrificing much recall.



Edges in the positive direction of the x-axis | Comparison with ground truth edges (all directions)

Figure 4.29 *– Edge detection results on the PB dataset. Left: detected edges in the positive x-direction overlaid on the point cloud. Right: comparison of positive x-direction detections (blue) against ground truth edges of all directions (red)*

Object reconstruction

Object reconstruction uses template matching to fill in incomplete objects detected by a prior erosion step. A partial match from the erosion, reconstructing the full object geometry even where portions of it are missing from the scan.

Input | Template Matched

Figure 4.30 *– Template matching applied to the PB dataset. Left: input with missing column data. Right: reconstructed points added by template matching shown in red.*

Figure 4.30 shows the result applied to the PB dataset, focusing on a specific column area. The input contains columns with significant missing data due to scan occlusion. The template matching correctly identifies the partially visible columns and reconstructs their full geometry

in red, inserting points where the scan has gaps. The same overshoot issue observed in the segmentation results is present here: columns that produce a stronger partial match receive a higher score and can be reconstructed slightly taller than their true extent. This is a direct consequence of the score threshold used to inform the reconstruction. A stricter threshold reduces overshoot but risks missing columns with less complete scan coverage.

Compared to surface reconstruction with a generic line or plane structuring element, template matching is better suited for filling geometrically complex holes that a simple structuring element would struggle to characterize correctly. A cylindrical column gap, for instance, has a spatial structure that a line structuring element cannot capture, whereas a structuring element of a sampled column matches the target geometry explicitly. This makes template matching a natural first step before surface reconstruction. Objects can be identified and completed individually, after which any remaining surface-level gaps are addressed globally. The main limitation is the requirement for a well-sampled reference structuring element and a reliable partial detection, which both depend on the quality and completeness of the available scan data.

5 Conclusion

5.1 Main Findings

The main research question of this thesis was defined as:

How can point-based mathematical morphology be enhanced and effectively applied to enable segmentation, object detection, and gap filling in heritage point clouds?

The answer is that point-based mathematical morphology, enhanced by a parallel implementation on the GPU and extended with the algorithm variants developed in this thesis, is fast, flexible, and sensitive enough to process real-world heritage point clouds. The technical contributions turn the proof-of-concept algorithms presented by Balado et al. (2020) into a tool capable of working effectively with heritage data. The use cases demonstrate that point-based mathematical morphology can produce useful results for segmentation, object detection, and gap filling on these datasets, without requiring any training data.

5.1.1 Technical contributions

The original sequential algorithms had severe limitations in processing speed. The erosion had a time complexity of $O(n \log n)$, and the dilation $O(n^2)$. This made both algorithms impractical on large input point clouds. Using the parallel GPU implementation, a time complexity of $O(n)$ was achieved for both. This resulted in a speed-up of approximately 100x for the erosion and enabled the dilation to handle input sizes that were nearly impossible before.

The binary output of the original erosion was poorly equipped to deal with real-world heritage data, as the noise in the input made structuring elements rarely match the input perfectly. The erosion score replaces this harsh binary output with a continuous value representing the percentage of points matched. This allows the output to be filtered by confidence instead of being either accepted or rejected completely. This makes the algorithm more robust on detailed and noisy inputs and allows for more flexibility during subsequent processing tasks.

The original method also required the structuring element to be axis-aligned, which is a significant limitation in heritage sites where the same object can appear in many different orientations. The rotation of the structuring element using a TNB matrix keeps the structuring element in a logical upwards position when aligning it with a single vector, such as a predetermined surface direction, or when it is being swept across different orientations during the

brute-force approach. By eliminating one axis of rotation, the structuring element can be applied effectively in multiple orientations, eliminating the axis alignment constraint.

Finally, the self-adaptive structuring elements address the density variation that is common in point clouds. Both a discrete and a continuous variant were implemented. During the dilation, they eliminate the harsh density borders and artificial densification, and during the erosion, they are able to more accurately find matches across varying local densities.

5.1.2 Heritage use cases

The segmentation results are promising. A basic implementation of a sequential segmentation workflow achieved a mIoU of 0.632 and an overall accuracy of 0.813, without any particular optimization of the structuring elements or processing order. The method requires no training data, and the structuring elements can be easily tailored to the geometric requirements of a specific dataset. This creates a strong starting point for semi-automatic heritage point cloud segmentation.

Object detection performed well, particularly on the Fontana dei Mesi dataset, where both the pillars and the shells were detected with an F1 score above 0.920. The opening and hit-or-miss transform, combined with the orientation-aware and density-aware variants, were capable of reliably detecting repeated geometries across a large and complex point cloud.

Gap filling produced more mixed results. Applied globally across an entire heritage point cloud using only a single structuring element, performance was limited. Due to the complexity of the input point cloud, the closing patched over gaps that were supposed to remain devoid of points. However, it was possible to accurately perform gap filling locally, indicating that more preprocessing steps are necessary. Edge detection, while achieving an F1 score of only 0.401 in the basic implementation, had false positives that were spatially close to the ground truth edges. This suggests that the detections are reasonable even where they do not exactly match the ground truth. Object reconstruction via template matching showed more consistent results and represents a more practical approach to gap filling for more complex objects.

5.2 Relevance

5.2.1 Addition of core processing techniques

Erosion and dilation are fundamental point cloud processing operations that remain difficult to perform reliably in conventional software. The GPU-accelerated implementations developed in this thesis, together with the compound operations they enable, add a set of fast and flexible geometric processing tools that can be applied to any 3D point cloud. By making creative use of these simple building blocks, even more complex compound operations can be created to geometrically process point clouds.

5.2.2 Interpretability and accessibility

Unlike the abstract input features of machine learning or the black box approach of deep learning, mathematical morphology makes its assumptions of the target geometry explicit via its structuring element. A user can easily inspect the geometry that the algorithms are looking

for and adjust as necessary to obtain the desired output. For heritage specialists who are not experts in the field of machine learning, this transparency makes it a more accessible and controllable method of processing heritage point clouds.

Furthermore, mathematical morphology is able to perform tasks without the need for ground truth classifications and annotated datasets. This makes it a suitable choice for smaller processing tasks, such as detecting a single object or filling a couple of specific gaps.

5.2.3 Towards machine learning on heritage data

One of the most important obstacles for machine learning and deep learning in the cultural heritage domain is the lack of labelled training data. Creating ground truth classifications and annotating datasets is a time-consuming process because automated processing is unable to handle the complexity and uniqueness of every individual heritage point cloud. By using the opening and hit-or-miss operations developed in this thesis, this process can be sped up by semi-automatically segmenting out objects like arches, columns, and vaults.

Furthermore, the output obtained from the erosion score can be used as a direct input feature for machine learning applications, such as random forest classification, alongside more traditional geometric measures such as sphericity and planarity. By creating more specific input features that are more closely aligned with the desired segmentation classes, the performance of machine learning segmentation can be increased.

5.2 Relevance

6

Discussion

6.1 Limitations

6.1.1 Lack of previous work

Point-based mathematical morphology is not a very explored research topic. Most of the related work has focused on surface-based morphological approaches or on entirely different methods for heritage point cloud processing. This leaves only the work of Balado et al. (2020) as the primary source for this thesis. This lack of prior work made it difficult to have a grounding in structuring element design, parameter selection, and processing workflows for heritage applications. Many of the design decisions made in this thesis were therefore based on empirical tests rather than on any previous findings. This raises the question of whether the chosen approaches are actually optimal.

6.1.2 Technical implementation

The parallel implementation developed in this thesis improved the performance compared to the baseline, but the code has not been optimized for more than what was needed to make the algorithms able to be evaluated. Non-core parts, in particular the sequential grouping algorithm, represent bottlenecks that a more experienced GPU programmer could likely address. Several features of the algorithms that would improve the practical usefulness of these tools were also not fully implemented. The erosion, for example, currently only applies the early exit when getting the binary output. The erosion score variant would also benefit from a hybrid that selects a specific score range to exit early when the score threshold becomes unattainable. This combination would keep the efficiency of early exiting while retaining the flexibility of the score. Similarly, the opening operation could be made more flexible by applying the erosion score through the dilation step. This would produce an opening score that reflects the presence of objects more reliably than the current binary output of the opening.

The TNB matrix rotation works well for objects that rotate around the vertical z-axis, but becomes less useful for objects appearing in different orientations on vertical surfaces, where the usage of the y-axis as a reference surface direction breaks down. This limits the use cases of the orientation-aware variants in practice.

The current user experience also presents an obstacle. Constructing structuring elements still requires several manual steps, and the workflow for compound operations introduces unnecessary overhead due to the repeated reading and writing of large point clouds to disk between processing stages. A more integrated workflow keeping point clouds in memory during operations would reduce this issue.

6.1.3 Data & resources

The segmentation use case was initially developed around using the ArCH dataset, which is a well-established dataset in heritage point cloud literature. This would have allowed for a more direct comparison with existing methods. The dataset was taken offline during the course of this thesis, which required switching to the Images&PointClouds Cultural Heritage Dataset. While this dataset provides ground truth labels for five heritage scenes, it contains a significant amount of noise and artificial reconstruction artifacts that affect segmentation quality in ways that are difficult to isolate from the performance of the algorithm itself.

In general, the availability of useful benchmarking datasets for heritage point cloud processing remains limited. For gap filling and density-aware object detection, no appropriate real-world benchmark existed, and therefore, required the construction of artificial test datasets. This limits the conclusions that can be drawn about real-world performance. The variety of acquisition methods represented in the experiments is also small. Object detection is only performed on LiDAR data and gap filling on synthetic photogrammetry derived data. This limits how broadly the results can be interpreted across all types of heritage data.

A more practical limitation that affected the scope of the experiments was the size of the datasets themselves. Many heritage point cloud datasets are distributed as large zip files of hundreds of gigabytes. The individual scans they contain are too large to load directly on the hardware available for this thesis. Only subsampled versions of the data were used for the heritage use cases. No preprocessing other than subsampling, such as denoising or outlier removal, was applied. The effect of such preprocessing on the algorithm's performance remains untested.

6.1.4 Validity & generalizability

The metrics used in this thesis are sensitive to several factors: the choice of structuring element, the threshold distance, the score value used to filter detections, the density of the input data, and the specific geometry of the heritage site. This sensitivity makes it difficult to draw strong conclusions from the numbers alone or to make meaningful comparisons with other methods, working under different conditions and on different datasets. The results should therefore be interpreted as indicative of what the method can achieve under the tested conditions, rather than as a definitive benchmark.

The generalizability of the approach beyond the datasets tested here is also uncertain. The method was developed and evaluated on a small number of heritage buildings and monuments, all acquired through terrestrial LiDAR or photogrammetry in relatively controlled conditions. Whether the structuring elements and processing workflows developed here would transfer to

more varied heritage types, such as ruins, wooden structures, or sites with irregular organic geometries, is to be seen. Extension beyond heritage into other domains, such as urban infrastructure or natural environments, is possible given that the method is geometry-based, but this has not been tested.

6.1.5 Heritage domain knowledge

The applications developed in this thesis were designed keeping the geometric requirements of heritage point clouds in mind. However, without a deep understanding of the cultural heritage domain, it is difficult to assess how well these tools align with the actual workflows and needs of heritage specialists. By working together more closely with people who work in the heritage field, new priorities and requirements could be brought to light. This could make it possible to extend mathematical morphology to new use cases and make it better suited to handle the ones currently explored.

6.2 Future Work

6.2.1 Further optimization

The current implementation uses OpenGL compute shaders, which were chosen in part for their integration with C++ environments and for their usage in CloudCompare. Alternative GPU programming frameworks, such as Vulkan or CUDA, offer even more low-level hardware access and could result in further performance improvements. Furthermore, a comparison between the spatial hash used throughout this thesis and other spatial data structures, such as octrees or kD-trees, would also be an interesting topic to explore. While the hash grid was selected based on its theoretical constant-time lookup, an actual comparison across a range of input densities and structuring element sizes would either validate this choice or identify conditions under which a different spatial structure performs better.

Compound operations currently require a significant overhead from repeatedly reading and writing large point clouds to disk between processing steps. Implementing a sequence of operations, which moves the output of one algorithm seamlessly to the input of the next without intermediate I/O, would reduce this bottleneck and make compound operations practical for interactive use. Further performance improvements could also be explored through more intentional GPU memory management, such as optimizing the data layout to improve cache coherence and reduce latency.

Finally, the algorithms still regularly hit the GPU limits as inputs become bigger. Integrating a better chunking strategy in which parts of the input are processed separately and stitched back together after could enable the processing of larger point clouds even on more limited hardware.

6.2.2 User friendliness

The current implementation implements eleven different erosion and dilation variants as separate algorithms. In practice, most of these could be unified into a single erosion and a single dilation function with optional parameters controlling orientation-awareness, density-

awareness, and score output. Combining the density and orientation variants into a single algorithm would further simplify the usage for an inexperienced user.

Structuring element construction is currently a mostly manual process. Several improvements would make this more efficient. Automatic centering of the structuring element on the origin, automatic alignment of its primary axis to the y-axis, for example, using PCA on the input points, and automatic generation of the negative structuring element required for the hit-or-miss transform would all reduce the number of manual steps involved. Object selection via edge detection could provide a more intuitive way of defining a structuring element directly from the point cloud, allowing the user to simply indicate one of the target objects as a reference rather than constructing the element from scratch.

The most practical improvement to user experience would be the development of a CloudCompare plugin. CloudCompare is already widely used in heritage point cloud processing, already uses OpenGL, and provides a graphical interface that would allow parameters to be set through a panel instead of through code. Default values for thresholds and structuring element density, calculated automatically from the properties of the input point cloud, would further lower the barrier to use for non-technical users.

6.2.3 Expanding the use cases

The current segmentation and gap-filling workflows require a significant amount of manual configuration steps for each new dataset. More automatic segmentation workflows, for example, automatically choosing the score filter value, would make the method more practical for regular use. A dedicated hole detection to local gap filling workflow, where edge detection first identifies the boundaries of missing regions, and gap filling is then applied only within those regions, would address the limitation identified in the results, where global gap filling on complex scenes performed poorly.

Other than heritage, the geometry-based nature of mathematical morphology makes it potentially useful to a range of other point cloud domains. Urban infrastructure, where objects such as lamp posts, curbs, and road markings have well-defined geometries, is a logical fit. The method may also generalize to less structured environments, though this would need to be verified. Other applications, such as point cloud simplification, skeletonization, and denoising, are also worth exploring, as these are established applications of mathematical morphology in image processing that have not yet been investigated using a point-based approach.

Bibliography

Ag2gaeh. (2008,). *Spherical coordinate system (local basis)*. https://commons.wikimedia.org/wiki/File:Kugelkoord-lokb-e.svg

Balado, J., Oosterom, P. van, Díaz-Vilariño, L., & Lorenzo, H. (2021). Point-based morphological opening with input data retrieval. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 53–58. https://doi.org/10.5194/isprs-annals-VIII-4-W2-2021-53-2021

Balado, J., Oosterom, P. van, Díaz-Vilariño, L., & Meijers, M. (2020). Mathematical morphology directly applied to point cloud data. *ISPRS Journal of Photogrammetry and Remote Sensing*, *168*, 208–220. https://doi.org/10.1016/j.isprsjprs.2020.08.011

Bentley, J. L. (1975b). *A Survey of Techniques for Fixed Radius Near Neighbor Searching* [Technical report]. https://doi.org/10.2172/1453938

Bentley, J. L. (1975a). Multidimensional binary search trees used for associative searching. *Commun. ACM*, *18*(9), 509–517. https://doi.org/10.1145/361002.361007

Calderon, S., & Boubekeur, T. (2014). Point morphology. *ACM Trans. Graph.*, *33*(4), 45:1–45:13. https://doi.org/10.1145/2601097.2601130

cchu79. (2023,). *Fibonacci Sphere [Answer]*. https://stackoverflow.com/a/76572678

Cera, V., Antuono, G., Campi, M., D'Agostino, P., Cera, V., Antuono, G., Campi, M., & D'Agostino, P. (2025). Data Quality, Semantics, and Classification Features: Assessment and Optimization of Supervised ML-AI Classification Approaches for Historical Heritage. *Heritage*, *8*(7). https://doi.org/10.3390/heritage8070265

CyArk. (2019,). *Palace of Fine Arts - Photogrammetry - Terrestrial, LiDAR - Terrestrial, Photogrammetry - Aerial*. Open Heritage 3D. https://doi.org/10.26301/vdae-mr89

de Vries, J. (2020,). *LearnOpenGL — Normal Mapping*. https://learnopengl.com/Advanced-Lighting/Normal-Mapping

Feng, Y., Xu, Y., Xia, Y., Brenner, C., & Sester, M. (2024). Gap completion in point cloud scene occluded by vehicles using SGC-Net. *ISPRS Journal of Photogrammetry and Remote Sensing*, *215*, 331–350. https://doi.org/https://doi.org/10.1016/j.isprsjprs.2024.07.009

Frías, E., Balado, J., Díaz-Vilariño, L., & Lorenzo, H. (2020). Point cloud room segmentation based on indoor spaces and 3D mathematical morphology. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 49–55. https://doi.org/10.5194/isprs-archives-XLIV-4-W1-2020-49-2020

Gil, A., Arayici, Y., Kumar, B., & Laing, R. (2024). Machine and Deep Learning Implementations for Heritage Building Information Modelling: A Critical Review of Theoretical and Applied Research. *J. Comput. Cult. Herit.*, *17*(3), 36:1–36:22. https://doi.org/10.1145/3649442

Green, S. (2010). *Particle Simulation using CUDA* [Technical report]. https://developer.download.nvidia.com/assets/cuda/files/particles.pdf

Grilli, E., Farella, E. M., Torresani, A., & Remondino, F. (2019). Geometric features analysis for the classification of cultural heritage point clouds. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 541–548. https://doi.org/10.5194/isprs-archives-XLII-2-W15-541-2019

Jones, M. T., & Plassmann, P. E. (1993). A Parallel Graph Coloring Heuristic. *SIAM Journal on Scientific Computing*, *14*(3), 654–669. https://doi.org/10.1137/0914041

Keinert, B., Innmann, M., Sänger, M., & Stamminger, M. (2015). *Spherical Fibonacci Mapping*. *34*(6). https://doi.org/10.1145/2816795.2818131

Khronos Group. (2012). *The OpenGL Graphics System: A Specification (Version 4.3 Core Profile)*. https://registry.khronos.org/OpenGL/specs/gl/glspec43.core.pdf

Kress Foundation. (2021,). *Houghton Hall - LiDAR - Terrestrial*. Open Heritage 3D. https://doi.org/10.26301/sjyw-3r43

Li, W., Pan, J., Hasegawa, K., Li, L., Tanaka, S., Li, W., Pan, J., Hasegawa, K., Li, L., & Tanaka, S. (2024). Missing Region Completion Network for Large-Scale Laser-Scanned Point Clouds: Application to Transparent Visualization of Cultural Heritage. *Remote Sensing*, *16*(15). https://doi.org/10.3390/rs16152758

Li, Y., Yong, B., Oosterom, P. V., Lemmens, M., Wu, H., Ren, L., Zheng, M., Zhou, J., Li, Y., Yong, B., Oosterom, P. V., Lemmens, M., Wu, H., Ren, L., Zheng, M., & Zhou, J. (2017). Airborne LiDAR Data Filtering Based on Geodesic Transformations of Mathematical Morphology. *Remote Sensing*, *9*(11). https://doi.org/10.3390/rs9111104

Lien, J.-M. (2008). Covering Minkowski sum boundary using points with applications. *Computer Aided Geometric Design*, *25*(8), 652–666. https://doi.org/10.1016/j.cagd.2008.06.006

Liu, Z., Oosterom, P. van, Balado, J., Swart, A., & Beers, B. (2022). Detection and reconstruction of static vehicle-related ground occlusions in point clouds from mobile laser scanning. *Automation in Construction*, *141*, 104461. https://doi.org/10.1016/j.autcon.2022.104461

Matrone, F., Grilli, E., Martini, M., Paolanti, M., Pierdicca, R., Remondino, F., Matrone, F., Grilli, E., Martini, M., Paolanti, M., Pierdicca, R., & Remondino, F. (2020). Comparing Machine and Deep Learning Methods for Large 3D Heritage Semantic Segmentation. *ISPRS International Journal of Geo-Information*, *9*(9). https://doi.org/10.3390/ijgi9090535

Moyano, J., León, J., Nieto-Julián, J. E., & Bruno, S. (2021). Semantic interpretation of architectural and archaeological geometries: Point cloud segmentation for HBIM parame-

terisation. *Automation in Construction*, *130*, 103856. https://doi.org/10.1016/j.autcon.2021.103856

Nebiker, S., Bleisch, S., & Christen, M. (2010). Rich point clouds in virtual globes – A new paradigm in city modeling?. *Computers, Environment and Urban Systems*, *34*(6), 508–517. https://doi.org/10.1016/j.compenvurbsys.2010.05.002

Pellis, E., Masiero, A., Betti, M., Tucci, G., & Grussenmeyer, P. (2025b). A Deep Learning Multiview Approach for the Semantic Segmentation of Heritage Building Point Clouds. *International Journal of Architectural Heritage*, *19*(12), 3117–3139. https://doi.org/10.1080/15583058.2025.2485242

Pellis, E., Masiero, A., Betti, M., Tucci, G., & Grussenmeyer, P. (2025a). A photogrammetric image-point dataset for the semantic segmentation of heritage buildings. *Data in Brief*, *60*, 111661. https://doi.org/https://doi.org/10.1016/j.dib.2025.111661

Pierdicca, R., Paolanti, M., Matrone, F., Martini, M., Morbidoni, C., Malinverni, E. S., Frontoni, E., Lingua, A. M., Pierdicca, R., Paolanti, M., Matrone, F., Martini, M., Morbidoni, C., Malinverni, E. S., Frontoni, E., & Lingua, A. M. (2020). Point Cloud Semantic Segmentation Using a Deep Learning Framework for Cultural Heritage. *Remote Sensing*, *12*(6). https://doi.org/10.3390/rs12061005

Previtali, M., Scaioni, M., Barazzetti, L., Brumana, R., & Roncoroni, F. (2013). Automated detection of repeated structures in building facades. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 241–246. https://doi.org/10.5194/isprsannals-II-5-W2-241-2013

Ren, Y., Chu, T., Jiao, Y., Zhou, M., Geng, G., Li, K., Cao, X., Ren, Y., Chu, T., Jiao, Y., Zhou, M., Geng, G., Li, K., & Cao, X. (2022). Multi-Scale Upsampling GAN Based Hole-Filling Framework for High-Quality 3D Cultural Heritage Artifacts. *Applied Sciences*, *12*(9). https://doi.org/10.3390/app12094581

Serra, J. P. (1982). *Image Analysis and Mathematical Morphology*. Academic Press. https://archive.org/details/imageanalysismat0000serr

Shah, H., Ghadai, S., Gamdha, D., Schuster, A., Thomas, I., Greiner, N., & Krishnamurthy, A. (2022). GPU-Accelerated Collision Analysis of Vehicles in a Point Cloud Environment. *IEEE Computer Graphics and Applications*, *42*(5), 37–50. https://doi.org/10.1109/MCG.2022.3177890

Tabib, R. A., Hegde, D., Anvekar, T., & Mudenagudi, U. (2023). *DeFi: Detection and Filling of Holes in Point Clouds Towards Restoration of Digitized Cultural Heritage Models*. 1603–1612. https://openaccess.thecvf.com/content/ICCV2023W/e-Heritage/html/Tabib_DeFi_Detection_and_Filling_of_Holes_in_Point_Clouds_Towards_ICCVW_2023_paper.html

Teppati Lose, L., Sammartano, G., Chiabrando, F., Spano, A., Politecnico di Torino Architecture and Design Department, & Spreafico, A. (2023,). *The Twelve Months Fountain of Valentino Park*. Open Heritage 3D. https://doi.org/10.26301/7ymp-ta22

Teschner, M., Heidelberger, B., Müller, M., Pomeranets, D., & Gross, M. (2003). Optimized Spatial Hashing for Collision Detection of Deformable Objects. *Vmv'03: Proceedings of the Vision, Modeling, Visualization*, *3*, . https://www.researchgate.net/publication/2909661_Optimized_Spatial_Hashing_for_Collision_Detection_of_Deformable_Objects

Yang, S., Hou, M., Li, S., Yang, S., Hou, M., & Li, S. (2023). Three-Dimensional Point Cloud Semantic Segmentation for Cultural Heritage: A Comprehensive Review. *Remote Sensing*, *15*(3). https://doi.org/10.3390/rs15030548

A

Technical Implementation

In this appendix, the technical implementation used in this thesis is discussed. It gives a short overview of the code, the data processing, and the setup of the heritage use cases. The full repository and accompanying read-me file can be found at:
https://github.com/Alcardens/Mathematical-morphology-point-cloud

A.1 Code development

A.1.1 Set-up

Hardware
The following hardware was used when running the algorithms to generate the benchmarks and perform the heritage use cases. All components are available to the average consumer:

Table 7 *– Hardware used for benchmarking and the heritage use case*

Component	Model	Spec	Cores
CPU	AMD Ryzen 7 5800H	3.20 Ghz	8
GPU	NVIDIA RTX 3070 Laptop	8 GB	5120
RAM	-	16 GB	-
SSD	-	1 TB	-

Required C++ Packages
The following C++ packages are required to successfully build the code in the repository. All packages were acquired with vcpkg:

CGAL OPENGL GLEW GFLW3 LASLIB TBB

Python libraries
The following Python libraries were used in the creation of figures and tables for the benchmarking and the heritage use cases:

PANDAS MATPLOTLIB PYCLOUDCOMPARE PLYFILE

To run PyCloudCompare, a valid installation of CloudCompare is also required.

A.1.2 Shaders

Compute shaders are defined as strings in C++ and compiled at runtime by the GPU driver. Each shader is stored as a named constant in shaders.h. To illustrate the implementation, the erosion shader will be presented:

```
inline const char* SRC_ERODE = R"GLSL(
```

Every shader begins by declaring its buffer bindings and uniform variables. Buffers are arrays of data residing in GPU memory. The read-only buffers carry inputs that do not change during the dispatch, while writable buffers receive the output. Uniforms are values that are set once from the CPU and remain constant for every invocation in the dispatch. The erosion shader declares five buffers and six uniforms:

```
layout(local_size_x = 256) in;

layout(std430, binding = 0) readonly buffer Cells    { uint cells[];   };
layout(std430, binding = 1) readonly buffer InputPts { vec4 in_pts[];  };
layout(std430, binding = 2) readonly buffer SE       { vec4 se_pts[];  };
layout(std430, binding = 3)          buffer OutPts   { vec4 out_pts[]; };
layout(std430, binding = 4)          buffer OutCount { uint out_count; };

uniform uint  u_table_size;
uniform float u_cell_size;
uniform float u_min_dist_sq;
uniform uint  u_num_input;
uniform uint  u_se_count;
uniform uint  u_max_output;
```

`local_size_x = 256` specifies the workgroup size; each dispatch launches groups of 256 invocations that execute the shader simultaneously. Binding 0 holds the spatial hash cell table, binding 1 the input point coordinates, and binding 2 the structuring element offsets, all read-only. Binding 3 is the output point buffer written by surviving threads, and binding 4 is an atomic counter tracking how many points have been written. The uniforms carry the hash table size, cell size, squared minimum distance threshold, the number of input points in the current chunk, the number of SE offsets, and the output buffer capacity.

The main function is then the direct GPU implementation of the erosion pseudocode defined in Appendix B, written in plain C++. Each invocation is assigned one input point by its global invocation ID. It checks all SE offsets against the spatial hash, and if all matches succeed, atomically claims a slot in the output buffer:

```
void main() {
    uint id = gl_GlobalInvocationID.x;
    if (id >= u_num_input) return;

    for (uint j = 0u; j < u_se_count; ++j) {
        if (!has_neighbor(p + se_pts[j].xyz))
            return;
    }

    uint slot = atomicAdd(out_count, 1u);
    if (slot < u_max_output)
        out_pts[slot] = vec4(p, 0.0);
}
```

`gl_GlobalInvocationID.x` gives each invocation a unique index across the entire dispatch. The guard `if (id >= u_num_input) return` handles the case where the number of invocations launched is not a multiple of 256 and thus allows excess invocations to exit immediately. The inner loop translates each SE offset to the input point's position and queries the spatial hash. The early return on the first failed match implements the early-exit strategy discussed in Section 3.1.3. If all SE offsets find a match, `atomicAdd` increments the shared counter and returns the previous value as a unique write slot, ensuring that no two threads write to the same position in the output buffer. The `w` component of the output `vec4` is set to `0.0` and repurposed by score variants to carry the match ratio.

A.1.3 Dispatch

The dispatch layer, defined in `point_cloud_erosion.h`, is responsible for allocating GPU buffers, uploading data, launching the shader, and reading results back to the CPU. Each operation is implemented as a class in a dedicated header and source file pair. The configuration parameters are collected in a struct that specifies buffer sizes and algorithm thresholds:

```
struct ErosionConfig {
    float    cell_size;
    float    min_dist;
    uint32_t table_size;
    uint32_t max_input;
    uint32_t max_output;
};
```

These values are computed in `erode.h` based on the input point cloud size and structuring element, and passed to the eroder at construction time so that all GPU buffers can be allocated once before any data is uploaded. Some functions are defined in the class: `set_structuring_element` uploads the SE to GPU memory, `erode` runs the full dispatch pipeline including hash construction, and `get_result` downloads and returns the surviving points:

```cpp
class PointCloudEroder {
public:
    explicit PointCloudEroder(const ErosionConfig& cfg);
    ~PointCloudEroder();

    void set_structuring_element(const std::vector<std::array<float, 3>>& se);
    uint32_t erode(const std::vector<std::array<float, 3>>& input);
    Point_set get_result() const;

    private:
    ErosionConfig m_cfg;
    uint32_t      m_result_count = 0;
    uint32_t      m_se_count     = 0;

    uint32_t dispatch_erode(const std::vector<std::array<float, 3>>& input,
                            GLuint prog);

    void build_input_hash(uint32_t point_count);

    GLuint m_ssbo_input_cells  = 0;
    GLuint m_ssbo_input_pts    = 0;
    GLuint m_ssbo_se           = 0;
    GLuint m_ssbo_out_pts      = 0;
    GLuint m_ssbo_out_count    = 0;

    GLuint m_prog_insert = 0;
    GLuint m_prog_erode  = 0;

    static GLuint compile_compute(const char* src);
};
```

The private members hold the OpenGL buffer object handles (`m_ssbo_*`) and compiled shader program handles (`m_prog_*`). Separating `m_prog_insert` from `m_prog_erode` allows the same hash insertion shader to be reused across all operations. The `build_input_hash` method dispatches the insert shader to populate the spatial hash from the full input, and `dispatch_erode` runs the erosion shader and reads back the output count. The high-level `erode.h` header wraps all of this into a single function call, computing appropriate buffer sizes before constructing the eroder.

A.1.4 Hash Grid

The spatial hash is implemented as its own self-contained class, `GPUSpatialHash`, shared by all morphological operations. Like the eroder, it is configured through a parameter struct:

```cpp
struct SpatialHashConfig {
    float    cell_size;
    float    min_dist;
    uint32_t table_size;
    uint32_t max_points;
};
```

The class manages two GPU buffers: the cell table (`m_ssbo_cells`) and the point coordinate array (`m_ssbo_points`). It exposes methods to insert points, query occupancy, and bind the buffers to specific binding slots for use by other shaders:

```
class GPUSpatialHash {
public:
    explicit GPUSpatialHash(const SpatialHashConfig& cfg);
    ~GPUSpatialHash();

    void bind(GLuint cells_binding, GLuint points_binding) const;

    uint32_t point_count() const { return m_point_count; }
    uint32_t remaining()   const { return m_cfg.max_points - m_point_count; }
    bool     is_full()     const { return m_point_count >= m_cfg.max_points; }

    uint32_t upload_points(const std::vector<std::array<float, 3>>& pts);
    std::vector<std::array<float, 3>> download_points() const;

    void reset();

private:
    void dispatch_insert(uint32_t start_offset, uint32_t count);

    SpatialHashConfig m_cfg;
    uint32_t          m_point_count = 0;

    GLuint m_ssbo_cells  = 0;
    GLuint m_ssbo_points = 0;
    GLuint m_prog_insert = 0;

    static GLuint compile_compute(const char* src);
};
```

`bind` makes the hash available to any subsequently dispatched shader by binding its two buffers to the specified slots, for example, the erosion shader expects the cell table at binding 0 and the point array at binding 1. Insertion is performed by `dispatch_insert`, which launches the insert shader on a range of points starting at `start_offset` in `m_ssbo_points`. The `upload_points` method copies point data to GPU memory and calls `dispatch_insert`, while `reset` clears the cell table back to its empty sentinel value and sets `m_point_count` to zero, allowing the hash to be reused across operations without reallocating GPU memory.

A.1.5 Python bindings

All operations are exposed to Python via pybind11 as a compiled module named `morphology`, compatible with Python 3.12. Each binding takes file paths to PLY or LAS input files, loads and processes the point clouds in C++, and returns the algorithm runtime in milliseconds. An optional output path argument triggers PLY export of the result. A headless OpenGL context is initialized once at module import time and shared across all subsequent calls, avoiding the overhead of context creation on each invocation.

A.1.6 Benchmarking

The benchmarks are run in a Python Notebook, using a simple for-loop, and are stored in a pandas DataFrame. The Python functions defined in the module return the processing time in milliseconds as their output. To minimize warm-up effects, the first few runs are discarded. However, the run order can still influence the outcome, with algorithms that are run earlier performing slightly slower than algorithms that are run later. Generally, the algorithms are run in the order from smallest to largest input. The writing of the output to disk is not included in the benchmark.

A.2 Data processing

A.2.1 Data pre-processing

The datasets from OpenHeritage3D were provided as collections of individual scans. These were first subsampled, merged, and then subsampled again until the total point count was manageable for the available hardware setup described in Table 7. The resulting clouds were then segmented to retain only the most relevant portions for the heritage applications evaluated in this thesis.

For the density-aware object detection use case, the Houghton Hall dataset was divided into five sections and subsampled at different densities ranging from 0.02 to 0.1 meter point spacing. This produced a single cloud with a controlled density gradient to evaluate the density-aware erosion's ability to adapt across varying local densities.

The PB dataset required a different preprocessing approach to simulate real acquisition conditions. Twelve scans from ground level and four from an aerial viewpoint were processed using hidden point filters, which remove points not visible from a given camera position. The filtered scans were then merged to produce a point cloud with realistic occlusions, mimicking the incomplete coverage typical of multi-scan heritage acquisitions.

The Images&PointClouds Cultural Heritage datasets were used largely as provided. Minor modifications were made to the 5_CB dataset. The points without a class label were removed, and some excess synthetically generated floor areas were trimmed to reduce the total point count and speed up the segmentation evaluation without affecting the relevant geometry.

A.2.2 Structuring element creation

The generation of synthetic structuring elements using helper functions has been discussed in Section 3.4. For applications requiring a structuring element derived directly from the point cloud, such as extracting a window frame or a pillar, the following procedure was used in CloudCompare.

The target geometry is first segmented from the point cloud using CloudCompare's segment tool. The segmented fragment is then translated so that the chosen reference point, typically the most representative or central point of the geometry, is placed at the origin. The cloud is subsampled to achieve the desired point spacing, and a second subsampling pass can be applied if a coarser element is needed, for example, to speed up the erosion part of the opening. For

density-aware use cases, the structuring element is either processed through the continuous density labelling algorithm or subsampled at multiple densities and merged into a single point cloud with the density property map attached. For orientation-aware use cases, the element is rotated so that the surface normal at the reference point is aligned with the y-axis, matching the convention expected by the TNB rotation matrix used in the shader.

A.3 Heritage use case set-up

A.3.1 Implementing Compound Operations

The compound operations were implemented as functions in the Python notebook, chaining together the developed morphological operations in the sequences described in Section 3.4.4. Since each operation reads from and writes to PLY files, the compound pipelines require intermediate I/O, and temporary files are written to disk between operations and read by the next step in the chain. While this adds some overhead, it keeps each operation self-contained and makes it straightforward to inspect intermediate results.

Score filtering was implemented using Plyfile, which reads the erosion score from the w component stored in the output PLY and filters points above or below a specified threshold. In some cases, the filtering was also performed visually in CloudCompare by inspecting the score scalar field and manually selecting an appropriate threshold based on the score distribution.

Normal re-interpolation after dilation was handled using PyCloudCompare, a Python wrapper that allows CloudCompare to be called in terminal mode from within a Python script. This avoids having to manually re-export and re-import files into CloudCompare for each intermediate step. Since CloudCompare's nearest-neighbor scalar field interpolation operates on scalar values rather than vector attributes, the surface normals must first be exported into three separate scalar fields representing the x, y, and z components, stored as integer values. Each scalar field is then interpolated independently using a single nearest-neighbor lookup from the original input cloud, after which the three fields are recombined into a normal vector for each point.

A.3.2 Metric Calculation

Detection and segmentation performance were evaluated using standard precision and recall metrics derived from the spatial overlap between the predicted output and the ground truth. Points in the intersection of the prediction and the ground truth — having a neighbor within the minimum distance threshold — were counted as true positives. False positives were obtained by subtracting the ground truth from the prediction, retaining predicted points with no nearby ground truth point. False negatives were obtained by subtracting the prediction from the ground truth, retaining ground truth points not covered by the prediction. For the evaluation of the surface reconstruction and edge detection, points already present in the original input data were excluded from both the prediction and the ground truth before computing the intersection, since the goal is to assess only the quality of the newly added geometry rather than the preservation of existing points.

A.3 Heritage use case set-up

B

Algorithms

This appendix presents the pseudocode for the developed algorithms. To keep it more compact, only the score variants are shown for the extended erosion algorithms. The link to the full repository can be found in Appendix A.

B.1 Erosion algorithms

Algorithm 1 – erode-old

```
1:  procedure ERODE-OLD(data, se, d_t)
2:      ▷ Build spatial index on input point cloud
3:      T ← KDTree(data)
4:
5:      result ← ∅
6:
7:      ▷ For each input point check if all translated SE offsets are covered
8:      for p ∈ data do
9:          ▷ Align SE anchor to p
10:         t ← p − se[0]
11:
12:         matched ← true
13:         for s ∈ se[1...] do
14:             s′ ← s + t
15:             d ← nn-dist(s′, T)
16:             if d > d_t then
17:                 matched ← false
18:                 BREAK
19:             end
20:         end
21:
22:         if matched then
23:             result ← result ∪ {p}
24:         end
25:     end
26:
27:     return result
28: end
```

Algorithm 2 – erode

```
1:  procedure ERODE(data, se, d_min)
2:      ▷ Build spatial hash from all input points with cell size d_min
3:      H ← SpatialHash(data, d_min)
4:
5:      result ← ∅
6:
7:      ▷ In parallel, every invocation handles one input point
8:      for p ∈ data do
9:          survived ← true
10:         for s ∈ se do
11:             ▷ Translate SE offset to point p
12:             s′ ← s + p
13:             ▷ Search 27 neighboring cells in H
14:             d ← nn-dist(s′, H)
15:             if d > d_min then
16:                 survived ← false
17:                 BREAK
18:             end
19:         end
20:         if survived then
21:             result ← result ∪ {p}
22:         end
23:     end
24:
25:     return result
26: end
```

Algorithm 3 – erode-score

```
1:  procedure ERODE-SCORE(data, se, d_min)
2:      ▷ Build spatial hash from all input points with cell size d_min
3:      H ← SpatialHash(data, d_min)
4:
5:      result ← ∅
6:
7:      ▷ In parallel, every invocation handles one input point
8:      for p ∈ data do
9:          matched ← 0
10:         for s ∈ se do
11:             ▷ Translate SE offset to point p
12:             s′ ← s + p
13:             ▷ Search 27 neighboring cells in H
14:             d ← nn-dist(s′, H)
15:             if d ≤ d_min then
16:                 matched ← matched + 1
17:             end
18:         end
19:
20:         ▷ Store score as w component of output point
21:         score ← matched / |se|
22:         result ← result ∪ {(p, score)}
23:     end
24:
25:     return result
26: end
```

Algorithm 4 – erode-density-score

```
1:  procedure ERODE-DENSITY-SCORE(data, se, Δ)
2:    ▷ Estimate local density for each point
3:    data ← density-estimate(data)
4:
5:    ▷ Find minimum distance
6:    d_min ← ∞
7:    for s ∈ se do
8:      d_min ← min(d_min, s.w)
9:    end
10:
11:   ▷ Build spatial hash with cell size d_min
12:   H ← SpatialHash(data, d_min)
13:
14:   result ← ∅
15:
16:   ▷ In parallel, every invocation handles one input point
17:   for p ∈ data do
18:     ▷ Clamp local density to valid range
19:     ρ ← clamp(p.w, ρ_min, ρ_max)
20:     d_sq ← ρ^2
21:     shells ← ⌈ρ/d_min⌉
22:
23:     matched ← 0
24:     applicable ← 0
25:     for s ∈ se do
26:       ▷ Skip SE points outside the applicable density band
27:       if Δ = 0 ∧ ρ > s.w then
28:         CONTINUE
29:       end
30:       if Δ > 0 ∧ |s.w − ρ| > Δ/2 then
31:         CONTINUE
32:       end
33:
34:       applicable ← applicable + 1
35:       s′ ← s.xyz + p.xyz
36:       if nn-dist(s′, H, d_sq, shells) ≤ √d_sq then
37:         matched ← matched + 1
38:       end
39:     end
40:
41:     ▷ Score over applicable SE points only, not total SE size
42:     if applicable > 0 then
43:       score ← matched / applicable
44:     end
45:     result ← result ∪ {(p, score)}
46:   end
47:
48:   return result
49: end
```

Algorithm 5 – erode-orientation-score

```
1:  procedure ERODE-ORIENTATION-SCORE(data, se, normals, d_min)
2:    ▷ Build spatial hash from all input points with cell size d_min
3:    H ← SpatialHash(data, d_min)
4:
5:    result ← ∅
6:
7:    ▷ In parallel, every invocation handles one input point
8:    for p ∈ data do
9:      ▷ Build TBN rotation matrix from surface normal
10:     n⃗ ← normals[p]
11:     R ← TNB(n⃗)
12:
13:     ▷ Count matched SE offsets after rotating into surface frame
14:     matched ← 0
15:     for s ∈ se do
16:       s′ ← p + R · s
17:       if nn-dist(s′, H) ≤ d_min then
18:         matched ← matched + 1
19:       end
20:     end
21:
22:     score ← matched / |se|
23:     ▷ Store score in w, normal in output normal buffer
24:     result ← result ∪ {(p, score, n⃗)}
25:   end
26:
27:   return result
28: end
```

Algorithm 6 – erode-orientation-score-brute

```
1:  procedure ERODE-ORIENTATION-SCORE-BRUTE(data, se, d_min, α)
2:    ▷ Build spatial hash from all input points with cell size d_min
3:    H ← SpatialHash(data, d_min)
4:
5:    ▷ Subsample structuring element
6:    se_p ← Subsample(se, d_min * 4)
7:
8:    result ← ∅
9:
10:   ▷ In parallel, every invocation handles one input point
11:   for p ∈ data do
12:     ▷ Phase 1 — coarse sweep with se_p
13:     best_θ ← 0
14:     best_φ ← π/2
15:     best_p ← −1
16:
17:     for θ ∈ [0, 2π) at step α do
18:       for φ ∈ [0, π] at step α do
19:         R ← TNB(dir(θ, φ))
20:         s_p ← score(se_p, p, R, H)
21:         if s_p > best_p then
22:           best_p ← s_p
23:           (best_θ, best_φ) ← (θ, φ)
24:         end
25:       end
26:     end
27:
28:     ▷ Phase 2 — erode with full se
29:     R ← TNB(dir(best_θ, best_φ))
30:     s ← score(se, p, R, H)
31:
32:     ▷ Store score and best-fit orientation as output normal
33:     n⃗ ← dir(best_θ, best_φ)
34:     result ← result ∪ {(p, s, n⃗)}
35:   end
36:
37:   return result
38: end
```

B.2 Dilation algorithms

Algorithm 7 – dilate-old

```
1:  procedure DILATE-OLD(data, se, d_t)
2:      ▷ Build spatial index on input point cloud
3:      T ← KDTree(data)
4:
5:      candidates ← ∅
6:
7:      ▷ For each input point translate SE and collect new candidates
8:      for p ∈ data do
9:          ▷ Align SE anchor to p
10:         t ← p − se[0]
11:
12:         for s ∈ se[1...] do
13:             s′ ← s + t
14:             d ← nn-dist(s′, T)
15:             if d > d_t then
16:                 candidates ← candidates ∪ {s′}
17:             end
18:         end
19:     end
20:
21:     ▷ Filter candidates: keep only points far enough from all accepted points
22:     filtered ← ∅
23:     for c ∈ candidates do
24:         T′ ← KDTree(filtered)
25:         if filtered = ∅ ∨ nn-dist(c, T′) > d_t then
26:             filtered ← filtered ∪ {c}
27:         end
28:     end
29:
30:     return data ∪ filtered
31: end
```

Algorithm 8 – dilate

```
1:  procedure DILATE(data, se, d_min)
2:      ▷ Split input into spatially separated groups so SE offsets cannot overlap
3:      G ← split(data, diameter(se) + d_min)
4:
5:      ▷ Insert all original points into hash before any group is processed
6:      H ← SpatialHash(data, d_min)
7:
8:      candidates ← ∅
9:
10:     ▷ Process each group sequentially
11:     for g ∈ G do
12:         ▷ In parallel, every invocation handles one candidate point
13:         for (p, s) ∈ g × se do
14:             ▷ Translate SE offset to input point
15:             c ← p + s
16:             ▷ Check if candidate is far enough from all accepted points in H
17:             if far-enough(c, H, d_min) then
18:                 candidates ← candidates ∪ {c}
19:             end
20:         end
21:
22:         ▷ Update H with accepted candidates before next group
23:         H ← H ∪ candidates
24:     end
25:
26:     return H
27: end
```

Algorithm 9 – dilate-density

```
1:  procedure DILATE-DENSITY(data, se, Δ)
2:    ▷ Estimate local density for each input point
3:    data ← density-estimate(data)
4:
5:    ▷ Split input into groups using density-scaled grouping distance
6:    G ← split(data, diameter(se) + ρ_min)
7:
8:    ▷ Insert all original points into hash before any group is processed
9:    H ← SpatialHash(data, ρ_min)
10:
11:   candidates ← ∅
12:
13:   for g ∈ G do
14:     ▷ In parallel, every invocation handles one candidate point
15:     for (p, s) ∈ g × se do
16:       ▷ Clamp local density and derive adaptive threshold
17:       ρ ← clamp(p.w, ρ_min, ρ_max)
18:
19:       ▷ Skip SE points outside the applicable density band
20:       if Δ = 0 ∧ ρ > s.w then
21:         continue
22:       end
23:       if Δ > 0 ∧ |s.w − ρ| > Δ/2 then
24:         continue
25:       end
26:
27:       c ← p.xyz + s.xyz
28:       shells ← ⌈ρ/d_min⌉
29:       if far-enough(c, H, ρ, shells) then
30:         candidates ← candidates ∪ {(c, ρ)}
31:       end
32:     end
33:
34:     H ← H ∪ candidates
35:   end
36:
37:   return H
38: end
```

Algorithm 10 – dilate-orientation

```
1:  procedure DILATE-ORIENTATION(data, se, normals, d_min)
2:    ▷ Split input into spatially separated groups
3:    G ← split(data, diameter(se) + d_min)
4:
5:    ▷ Insert all original points into hash before any group is processed
6:    H ← SpatialHash(data, d_min)
7:
8:    candidates ← ∅
9:
10:   for g ∈ G do
11:     ▷ In parallel, every invocation handles one candidate point
12:     for (p, s) ∈ g × se do
13:       ▷ Build TNB rotation matrix from surface normal of p
14:       n⃗ ← normals[p]
15:       R ← TNB(n⃗)
16:       ▷ Rotate SE offset into surface frame before translating
17:       c ← p + R · s
18:       if far-enough(c, H, d_min) then
19:         candidates ← candidates ∪ {c}
20:       end
21:     end
22:
23:     H ← H ∪ candidates
24:   end
25:
26:   return H
27: end
```

B.3 Helper algorithms

Algorithm 11 – add

```
1:  procedure ADD(original, addition, d_min)
2:      ▷ Hash all points from original
3:      H ← SpatialHash(original, d_min)
4:
5:      result ← original
6:
7:      ▷ In parallel, check each addition point against original
8:      for p ∈ addition do
9:          ▷ Accept only points not already covered by original
10:         if far-enough(p, H, d_min) then
11:             result ← result ∪ {p}
12:         end
13:     end
14:
15:     return result
16: end
```

Algorithm 12 – subtract

```
1:  procedure SUBTRACT(original, subtraction, d_min)
2:      ▷ Hash all points from subtraction cloud
3:      H ← SpatialHash(subtraction, d_min)
4:
5:      result ← ∅
6:
7:      ▷ In parallel, check each original point against subtraction
8:      for p ∈ original do
9:          ▷ Keep only points with no neighbour in subtraction
10:         if ¬ has-neighbor(p, H, d_min) then
11:             result ← result ∪ {p}
12:         end
13:     end
14:
15:     return result
16: end
```

Algorithm 13 – intersect

```
1:  procedure INTERSECT(original, intersection, d_min)
2:      ▷ Hash all points from intersection cloud
3:      H ← SpatialHash(intersection, d_min)
4:
5:      result ← ∅
6:
7:      ▷ In parallel, check each original point against intersection
8:      for p ∈ original do
9:          ▷ Keep only points that have a neighbour in intersection
10:         if has-neighbor(p, H, d_min) then
11:             result ← result ∪ {p}
12:         end
13:     end
14:
15:     return result
16: end
```

B.3 Helper algorithms

Declaration of AI usage

Parts of this work have benefited from the use AI/LLM, mainly to speed up tedious tasks, such as debugging and proofreading. The following tools have been used:

NotebookLM
- Retrieving specific content from papers during the literature review
- Relating different sources to each other

Claude
- Debugging of code and setting up OpenGL shaders in CLion
- Proofreading and suggesting changes related to structure and phrasing
- Generating typst tables and matplotlib graph layouts

Grammarly
- Correcting final spelling and grammar mistakes

All results, analyses, and conclusions have been created by me; the text has been refined using AI/LLM, but no new ideas have been inserted in the process. Sources have been personally consulted to verify their validity, and all code has been reviewed to work as intended. The content has been verified to be original and meet academic integrity standards.

Enhancing point-based mathematical morphology for processing applications in heritage point clouds