CGGS CW1: Geometry

s2693586

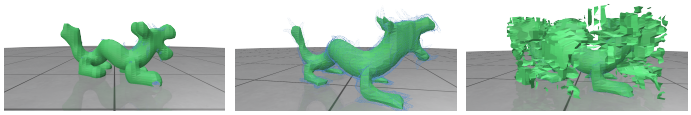
February 25, 2025

1 Section 1

1.1 Introduction

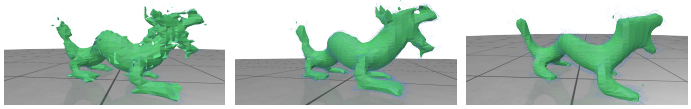
Moving Least Squares (MLS) is a technique for reconstructing smooth surfaces from point clouds. In this section, we will explore how parameters like polynomial degree N, influence radius h, off-surface displacement ϵ, and grid resolution affect reconstruction accuracy and efficiency. We will also compare different weight functions and analyse their impacts.

1.2 Parameter Exploration & Effects



(a) $N = 0$, 6.5s (b) $N = 1$, 13s (c) $N = 2$, 25s

Figure 1: Comparison of different Polynomial Degree (N); Grid = 32, $\epsilon = 0.01$, $h = 0.1$,

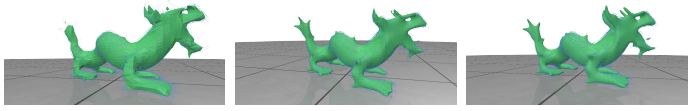


(a) $h = 0.01$, 13s (b) $h = 0.05$, 6.5s (c) $h = 0.15$, 28.9s

Figure 2: Comparison of different Influence Radius (h); Grid = 32, $\epsilon = 0.01$, $N = 1$

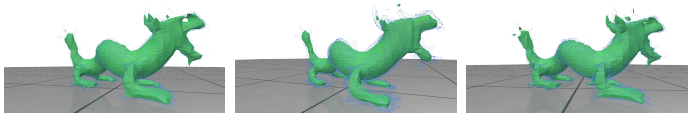
(a) $\epsilon = 1e-7$, 6.5s (b) $\epsilon = 0.01$, 6.5s (c) $\epsilon = 0.1$, 6.7s

Figure 3: Comparison of different ϵ; Grid = 32, $N = 1$, $h = 0.05$



(a) Grid = 32, 6.5s (b) Grid = 64, 52s (c) Grid = 128, 419s

Figure 4: Comparison of different Grid Resolutions; $\epsilon = 0.01$, $N = 1$, $h = 0.05$,



(a) Wendland (b) Gaussian (c) Singular

Figure 5: Comparison of different Weight Functions; Grid = 32, $\epsilon = 0.01$, $N = 1$, $h = 0.05$

1.3 Discussions

1. **Polynomial Degree (N)**: As we can see from Figure 1, when $N = 0$ (constant fit), it produces a rough approximation where the outcome is super simple and computationally cheap. When $N = 1$ (linear fit), it can capture more details compared to when $N = 0$, and it has a higher computational cost. When $N = 2$ (quadratic fit), overfitting and artifacts start to appear, as it is more sensitive to noise.

   The choice of N should depend on the level of detail in the input data. Small N can react well to data with noise but produce a rough approximation of the cloud points, while bigger N can approximate sharp details but is highly sensitive to noise. Higher N will also lead to higher computational costs. In our case, $N = 1$ produces the best result; therefore, it will be used in the other experiments.

2. **Influence Radius (h)**: From Figure 2, we can see that for lower h, the surface shows fine details, whereas higher h will lead to a smoother surface.

   Lower h is more sensitive to noise, and as it gets smaller, gaps might appear; higher h leads to loss of details and is also high in computational cost. $h = 0.05$ will be used in other experiments as it produces the best result.

3. **Epsilon** ϵ: From Figure 3, we can see that small ϵ is precise where the off-surface points are very close to the original points. It can preserve fine details, but it is sensitive to noise if the cloud points are small (the bunny only has 500 points, whereas the dragon has 25000 points). For large ϵ, the off-surface points are too far, which can lead to over-connected structures.

   The choice of ϵ will mainly depend on the size of the data. A moderate ϵ of 0.01 will be used for the dragon as it produces the best result in our cases.

4. **Grid Resolutions**: Figure 4 shows that grid resolution directly affects the quality of the output as well as computation cost. For small resolutions, details are lost, but the render time is short, whereas for higher resolutions, the time for rendering grows exponentially. However, the finest details are captured.

   Small grid resolution can be used for quick previews and large-scale reconstructions. Middle-size grid resolution will balance speed and detail well. High resolution should be used if details are important and if computational resources, especially time, are not a constraint.

5. **Weight Functions**: Wendland weight function gives zero weight beyond a radius h; this will avoid unwanted artefacts

as well as lead to a smooth surface and preserve as many details as possible. Gaussian weight function will assign non-zero weight to all points; it can have a strong smoothing effect, so it is useful in denoising. However, it may over-smooth the surface shown in Figure 5 (b). From Figure 5 (c), we can see that the singular weight function is good at preserving edges; it does less smoothing than Wendland and Gaussian, which makes it better for sharp features. However, it might also capture noises as it does not average neighbouring points.

By comparing all figures in Figure 5, we can see that Wendland does a relatively good job of combining the benefits of Gaussian and Singular weight functions. This explains why the Wendland weight function is being used in most cases of general MLS reconstructions.

1.4 Conclusion

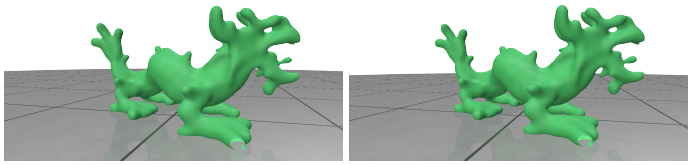
The selection of parameters for MLS reconstruction needs to balance detail preservation and noise sensitivity and, at the same time, be computationally efficient and will differ between different cloud points based on their size. In our case, for the *dragon-25000.off*, a polynomial degree of $N = 1$ offers a good trade-off between simplicity and accuracy. A radius $h = 0.05$ preserves details as well as avoids excessive smoothing. A ϵ of 0.01 prevents overconnected structures and preserves fine details. Grid resolution should be adjusted based on the computational resources available or the level of detail that is needed. Among the three weight functions we experiment with, Wendland provides the best balance between smoothing and feature preservation; this allows it to be the most suitable choice for general MLS reconstructions.

2 Section 2

2.1 Function Blending MLS

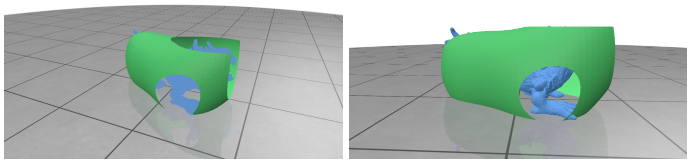
Compared to the Polynomial Scalar Moving Least Squares (PS-MLS), the Function-blending MLS (FB-MLS) defines distance functions per point and blends them using Wendland weights. This method is equivalent to PS-MLS with $N = 0$ and avoids solving a linear system, so it would be more efficient but more likely to be less accurate. In this section, we will compare the performance of the two methods, PS-MLS and FB-MLS and discuss the differences between them. For Section 2.2, all outcomes produced by the PS-MLS function will have a parameter of $N = 0$, and a small ϵ of 0.005 will be used as if ϵ is 0, it might cause instability.

2.2 Effects



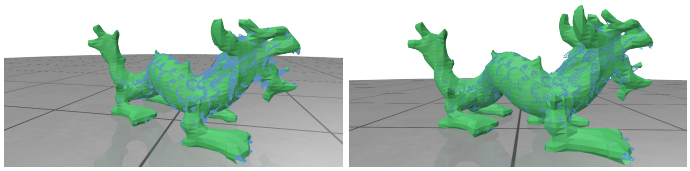
(a) PS-MLS, 376s (b) FB-MLS, 93s

Figure 6: PS-MLS and FB-MLS at 128 GridRes, $h = 0.05$



(a) PS-MLS, 135s (b) FB-MLS, 5.9s

Figure 7: PS-MLS and FB-MLS at 32 GridRes, $h = 0.5$



(a) PS-MLS, 5.8s (b) FB-MLS, 1.5s

Figure 8: PS-MLS and FB-MLS at 32 GridRes, $h = 0.05$

2.3 Disscussions

1. **Surface Quality**: At $N = 0$, the surface quality of the image produced by the original PS-MLS is similar to the one produced by FB-MLS. However, as mentioned in Section 1.3, we can increase N to preserve more details, whereas for FB-MLS, there is no such option N, so PS-MLS have the ability to preserve details better while FB-MLS might oversmooth.
2. **Rendering speed**: From Section 2.2, we can see that FB-MLS is significantly faster than PS-MLS as it avoids solving a least-squares system; this makes it computationally much more efficient. The speedup should be more noticeable in high resolutions and large point clouds.
3. **Effect of** h: From Figure 7 and Figure 8, we can see that both method behave similarly for different h ($N = 0$ for PS-MLS). For PS-MLS, we can see from Section 1.3 that at higher N, it will behave differently at higher N as PS-MLS interacts with h more strongly as N increases; this leads to a better curvature approximation.

2.4 Conclusion

Overall, FB-MLS and PS-MLS behave similarly when $N = 0$ for PS-MLS. However, PS-MLS has more flexibility, as it can use high-degree polynomials. This enables PS-MLS to preserve details, but the computational complexity increases significantly. The choice between FB-MLS and PS-MLS should depend on the application. FB-MLS is suitable for fast, smooth reconstructions, while PS-MLS is suitable for fine detail preservation, and more computation resources will be needed.

3 Section 3

3.1 Introduction

As we have identified in the previous sections, the PS-MLS is extremely inefficient as it computes local approximations by iterating over every grid point and over every point in the point cloud in each grid point. This double loop approach leads to an $O(n_{grid} \times n_{points})$ time complexity. When we come across large

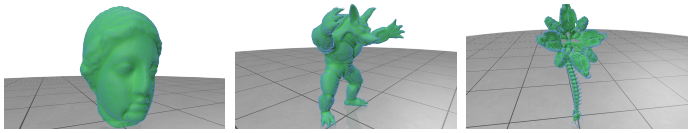
data sets (e.g. 100k+ points and a grid resolution of 128^3), the performance bottleneck will be severe due to the growth in computational complexity. As shown in Table 1, the average brute-force time for rendering a model with about 134K data points will be over 30 minutes, this makes the PS-MLS approach impractical for large-scale point clouds and fine grid resolutions.

3.2 The kd-tree Approach

The algorithm's bottleneck is mainly the repeated evaluation of the distance between the grid points and all point cloud samples. To address this issue, an efficient spatial lookup will be needed. A kd-tree (k-dimensional tree) is effective for nearest neighbour searches as it is a data structure based on spatial partitioning for organising points in k-dimensional space. kd-tree splits the data points along the dimension with the largest variance and organises them into a binary tree. This structure gives it an average complexity of $O(log(n))$ in radius searches and a worst-case complexity of $O(n)$. [1][2]

3.3 Evaluation

After implementing the kd-tree optimisation from [3] on top of our brute force approach, we evaluated the performance by comparing it with the brute-force approach we had in Section 1. We measured the average execution time for 3 large-scale point cloud models obtained from [4], namely Igea, Armadillo and Quad. They contain between 134k and 173k points and are shown in Figure 9.



(a) Igea (b) Armadillo (c) Quad

Figure 9: Cloud point models used in evaluation

Model	Points	Optimized Time	Brute-force Time
Igea	134k	29.5s	1032s
Armadillo	173k	47.8s	1652s
Quad	163k	68.2s	1736s

Table 1: Comparison of execution times between the optimized kd-tree-based implementation and the brute-force approach for different 3D point cloud models with 100k+ point cloud. The kd-tree approach significantly reduces computational time, making the algorithm scalable for large datasets.

Table 1 presents the execution times for both the optimised and brute-force approaches. The result has shown a significant speed-up:

- Igea: Reduced from 1032s to 29.5s, about 35x speed-up.
- Armadillo: Reduced from 1653s to 47.8s, about 35x speed-up.
- Quad: Reduced from 1736s to 68.2s, about 25x speed-up.

These improvements have indicated that the kd-tree has decreased the computation time significantly. The Quad has a smaller speed-up (25x); this is likely due to the structure and spatial distribution; as we can see from Figure 9, the quad model has more scattered and non-uniform point distribution (stem and leaves).

3.4 Conclusion

By introducing kd-tree for spatial query optimization, we significantly improve the computational efficiency of the PS-MLS algorithm. Compared to the traditional brute-force search method, kd-tree allows for faster neighbourhood search and decreases the computational complexity from $O(n_{grid} \times n_{points})$ to $O(log(n))$ (average case), which greatly reduces the computation time. Experimental results show that on three large-scale point cloud datasets, namely Igea, Armadillo and Quad, the computational speed of the kd-tree version is improved by 25× to 35×, which dramatically improves the scalability of the algorithm.

In addition, the Quad model's kd-tree speedup is not as significant as that of Igea and Armadillo due to its sparse point cloud distribution and more elongated structures. This shows that the efficiency of the kd-tree depends on the spatial distribution of the point cloud to a certain extent.

Overall, the kd-tree optimization makes PS-MLS more practical for processing large-scale point clouds and is suitable for real-world applications such as computer graphics, geometric modelling, and 3D reconstruction.

References

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[2] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[3] J. L. Blanco, "nanoflann: A c++ header-only library for nearest neighbor search with kd-trees," https://github.com/jlblancoc/nanoflann, 2014.

[4] A. Jacobson, "Common 3d test models," https://github.com/alecjacobson/common-3d-test-models, 2024, accessed: 2025-02-24.