Topic: A 3D Poisson surface reconstruction implementation

HUGO RIFFAUD DE TURCKHEIM, Sorbonne University

This paper presents a comprehensive report on the implementation of a *3D Poisson surface reconstruction algorithm*, conducted as part of the "IG3DA" course at Telecom Paris. It details the algorithmic steps undertaken, the reasons behind key implementation decisions, and evaluates their effectiveness and limitations. By critically analyzing these choices, the document aims understand the complexities of 3D surface reconstruction, offering insights into the trade-offs involved in algorithm design and application. This work contributes to a deeper understanding of Poisson-based reconstruction methods, highlighting potential areas for further research and optimization of the code provided.

1 INTRODUCTION

The 3D Poisson Surface Reconstruction is a sophisticated algorithm designed to create a 3D mesh from a cloud of points. Each point in the cloud, denoted as a sample s, is characterized by two main attributes: its spatial coordinates $s.p$ and its normal vector $s.\vec{N}$. The algorithm go through 4 key steps:

(1) Populating an octree with the point samples.
(2) Computing a vector field $\vec{V}$.
(3) Solving a pertinent linear system.
(4) Extracting an iso-surface to construct the 3D mesh.

The subsequent sections will discuss about these steps, providing a comprehensive walk through of the process.

2 SELECTION OF THE PROGRAMMING FRAMEWORK

Given the prevalence of C++ in computer graphics, the choice to implement this project in Python, specifically utilizing PyTorch, might seem unconventional. This decision was driven by several factors:

2.1 Advantages:

- **Efficient Parallel Operations**: PyTorch excels at handling matrix and array operations in parallel, significantly enhancing computational efficiency. A pivotal factor in this choice was PyTorch's seamless integration with GPU computing, vital for handling the intensive computations involved in 3D reconstruction.
- **Debugging and Visualization**: The ease of debugging and visualizing processes in PyTorch, aided by libraries like Matplotlib, allows an easier identification and resolution of issues within the code.

2.2 Disadvantages:

- **Performance Considerations**: Python's inherent performance limitations, particularly with for loops, pose a challenge. This necessitates a strategic approach to fully leverage PyTorch's capabilities for efficient array operations, avoiding the slow execution times associated with iterative loops.

This report is submitted as a part of project for Advanced 3D Computer Graphics (IMA904/IG3DA), Telecom Paris.

The original work is introduced by [Kazhdan et al. 2006].

3 POPULATING AN OCTREE

The initial phase involves distributing all data samples within an octree. This structure is constructed recursively, starting with a cube that encompasses all points, representing the octree's zero depth level. This cube is then subdivided into eight equal-volume nodes, incrementing the depth (D) by one, and this process is repeated for each node until each contains a single sample or a predefined depth is reached. This ensures every sample is located within a leaf node, optimizing the organization of the point cloud for subsequent steps.

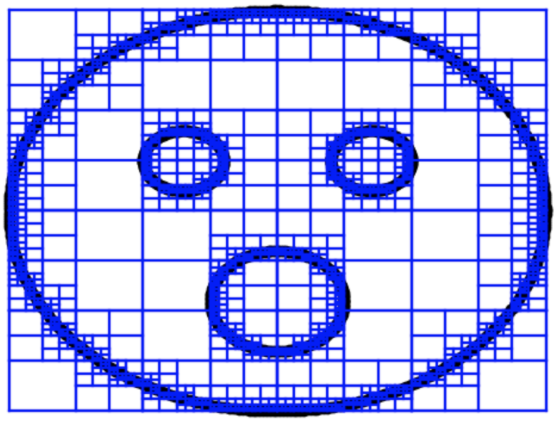

Fig. 1. A quadtree, the 2D version of an octree.

3.1 Computing the vector field $\vec{V}$

To compute the vector field $\vec{V}$, we apply the following formula:

$$\vec{V}(q) \equiv \sum_{s \in S} \sum_{o \in \mathrm{Nbgr}_d(s)} \alpha_{o,s} F_o(q)\ s.\vec{N}$$

This computation involves:

- Assigning a Gaussian function F_o to each tree node, facilitating the interaction between nodes and the vector field.
- Utilizing $\alpha_{o,s}$ as trilinear interpolation weights to finely tune the contributions of surrounding nodes.

In this formulation, the vector field $\vec{V}$ is represented as a linear sum of F_o. This implies that if we want to correctly estimate $\vec{V}$ using an octree of depth D we have to sample the space taken by the octree using a grid of size at least: 8^D or $2^D \times 2^D \times 2^D$. Instead of naively directly computing this formula for each point of the grid, we can remark a few things to make the estimation of $\vec{V}$ faster without losing information.

- **Selective Sampling**: Given F_o's Gaussian nature, distant points contribute negligibly to $\vec{V}$. Therefore, computations focus on samples near each point q, utilizing the octree's structure to limit calculations to relevant nodes.
- **Efficient Node Updating**: The octree also helps identify and update only those nodes containing samples or within close proximity to them, avoiding unnecessary calculations.
- **Batch Processing**: The independence of each sample's computation allows for processing in batches, reducing the need for loading all samples simultaneously.

By incorporating these strategies, we've successfully vectorized the entire computation process, minimizing the need for iterative loops and significantly enhancing efficiency.

Fig. 2. A slice of the vector field associated to the dragon model at depth $D = 8$.

4 LINEAR SYSTEM SOLVING

Upon acquiring the vector field $\vec{V}$, the next step involves solving a linear system represented as $L \cdot x = v$, where:

- $L_{o,o'} = \left\langle \frac{\partial^2 F_o}{\partial x^2}, F_{o'} \right\rangle + \left\langle \frac{\partial^2 F_o}{\partial y^2}, F_{o'} \right\rangle + \left\langle \frac{\partial^2 F_o}{\partial z^2}, F_{o'} \right\rangle$, encapsulates the Laplacian's interaction between basis functions across the domain.
- $v_o = \left\langle \nabla \cdot \vec{V}, F_o \right\rangle$, represents the divergence of $\vec{V}$ projected onto the basis functions.

The standard inner product $\langle \cdot, \cdot \rangle$ on function spaces is defined as:

$$\langle F, G \rangle = \int F(p) \cdot G(p)\, dp,$$

for scalar functions and

$$\langle \vec{U}, \vec{V} \rangle = \int_{[0,1]^3} \vec{U}(p) \cdot \vec{V}(p)\, dp,$$

for vector fields, facilitating the computation of both L and v.

4.1 Computation Challenges and Strategies

The computation of L presents notable challenges due to its dimensions ($8^D \times 8^D$) and sparsity. Addressing these, we employed two strategies:

(1) **Naive Approach**: Initially, a direct computation was adopted for its simplicity and to validate the implementation. Given the prohibitive expense of calculating all inner products, optimizations were introduced:
   - Limiting computations to nodes within a predefined distance threshold, significantly reducing the computational load.
   - Exploiting the matrix's sparsity and symmetry to compute only a portion of L, effectively halving the computational effort.
   - Recognizing and leveraging the repeating pattern in L to further optimize computations.

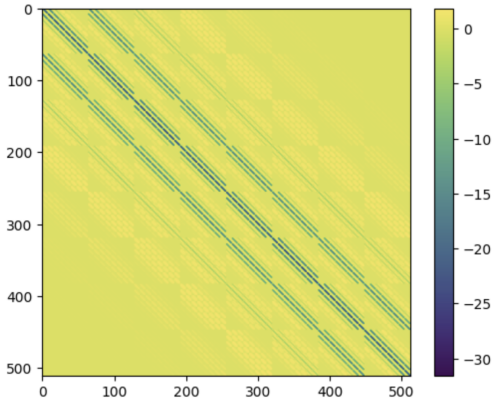

Fig. 3. Illustration of the repeating pattern in L with increasing depth D.

(2) **Refined Approach: Cascadic Poisson Solver**: To circumvent inefficiencies, we shifted to a cascading solver that iteratively refines the solution from coarse to fine resolutions. This approach selectively incorporates nodes based on their proximity to data points, enhancing computational efficiency and focusing resources on relevant portions of the domain.

ALGORITHM 1: Cascadic Poisson Solver [Kazhdan and Hoppe 2013]

for $d \in \{0, \ldots, D\}$ **do**
  `/* Iterate from coarse to fine */`
  for $d' \in \{0, \ldots, d-1\}$ **do**
    `/* Remove the constraints met at coarser depths */`
    $b^d = b^d - A^{dd'} x^{d'}$
  end
  `/* Adjust the system at depth d */`
  Relax $A^d x^d = b^d$
end

4.2 Algorithmic Details

The Cascadic Poisson Solver algorithm iteratively adjusts constraints and refines the solution across different depths, optimizing the linear system's resolution by progressively incorporating finer details. It needs more iterations of the solver to find a good solution at each depth but needs less space in memory, making it a valuable method.

Conjugate Gradient Method: Chosen for its efficiency in sparse systems, the conjugate gradient method provided the requisite precision and convergence speed. While alternative solvers like SGD or LBFGS were considered, the conjugate gradient's suitability for our sparse, symmetric system made it the preferred choice.

5 FINAL STEP: ISO-SURFACE EXTRACTION

The last step of our algorithm is the iso-surface extraction. Once the linear system has been solved, we obtain an indicator function which can be computed at each point of the grid using this formula:

$$\tilde{x} = \sum_o x_o F_o$$

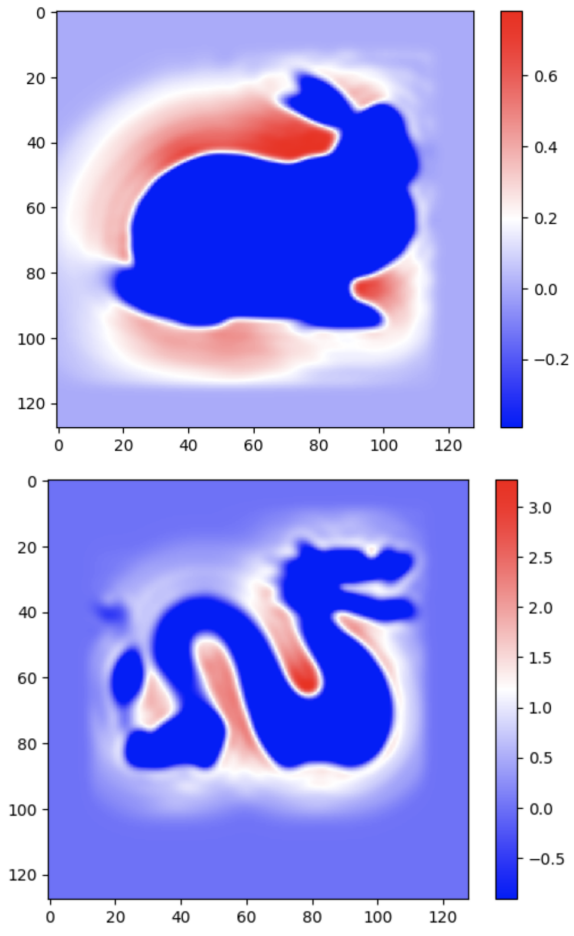

Fig. 4. Indicator functions obtained at $D = 6$.

To extract an isovalue, we used the formula provided in the original paper, which is simply the average value of the iso-surface at each sample position s:

$$\partial\tilde{M} = \{q \in \mathbb{R}^3 \mid \tilde{x}(q) = \gamma\} \text{ with } \gamma = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \tilde{x}(s \cdot p)$$

Using the marching cubes algorithm, it is possible to reconstruct the 3D mesh. In our implementation, we utilize the code provided by the scipy library to perform this operation. Additionally, we employed an external library to export the result in a .obj file format.

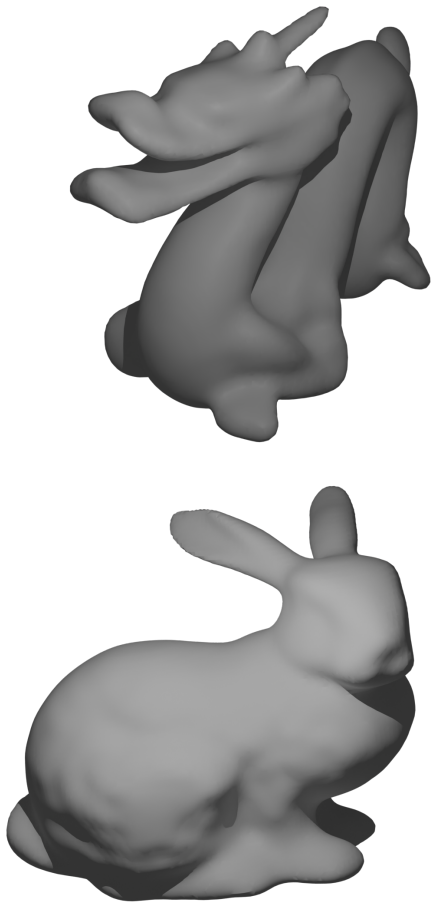

Fig. 5. Reconstructions using an octree of depth $D = 6$. The point clouds used are the points contained in the reconstructions available in the Stanford repository.

6 CONCLUSION

Through diligent implementation and rigorous testing, we have successfully transformed scattered point data into coherent 3D meshes. Despite the challenges faced, the outcomes underscore the algorithm's potential in various applications, from digital heritage preservation to biomedical imaging.

However, the journey to refine and optimize this implementation is far from complete. Future endeavors should focus on the following strategic enhancements:

(1) The adoption of basis functions with analytical solutions for inner product calculations could dramatically increase computational efficiency. This adjustment promises to extend the algorithm's applicability to higher-resolution data by mitigating current limitations on octree depth.
(2) Enhancing the robustness of iso-surface extraction against noise is crucial. Exploring adaptive thresholding techniques or incorporating machine learning-based noise discrimination could offer substantial improvements in handling real-world data variability.

REFERENCES

Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. 2006. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, Vol. 7. 0.

Michael Kazhdan and Hugues Hoppe. 2013. Screened poisson surface reconstruction. *ACM Transactions on Graphics (ToG)* 32, 3 (2013), 1–13.

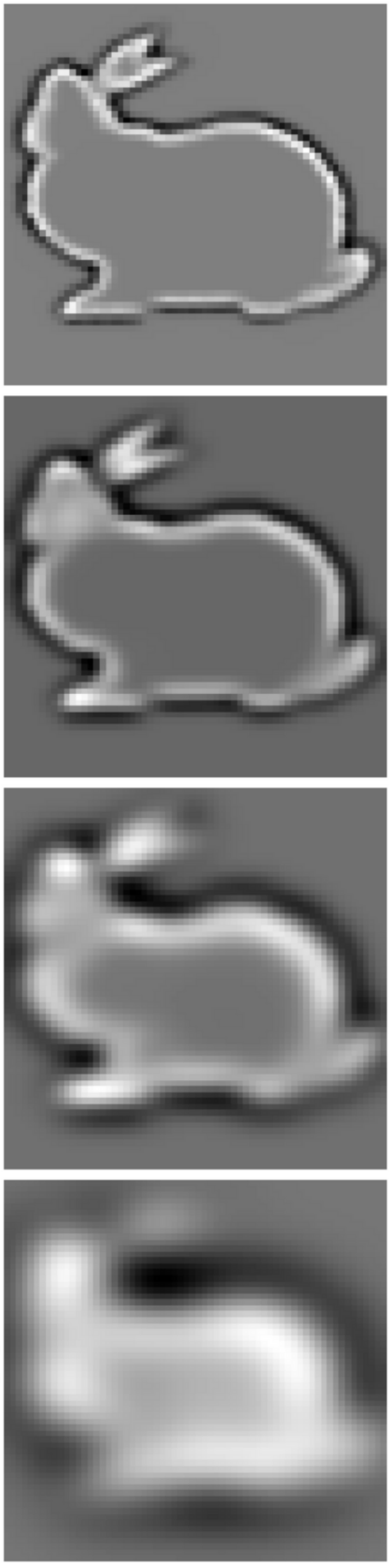

Fig. 6. Effect of the depth of the octree on the quality of the estimation of $\vec{V}$. From top to bottom, $D = [6, 5, 4, 3]$