# CS284A Final Report

I-LUN TSAI, University of California, Berkeley, USA
KEVIN XIONG, University of California, Berkeley, USA
SEAN WANG, University of California, Berkeley, USA
SHU-PING CHEN, University of California, Berkeley, USA

Our project is about rendering a mesh using the Ball-Pivoting Algorithm on a data set. For our Milestone, we focused on how the algorithm works when used on data pulled from the Stanford 3D Scanning Repository. Then, we attempted our own implementation to be able to use the algorithm ourselves.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: datasets, neural networks, gaze detection, text tagging

**ACM Reference Format:**
I-Lun Tsai, Kevin Xiong, Sean Wang, and Shu-Ping Chen. 2018. CS284A Final Report. *ACM Trans. Graph.* 37, 4, Article 111 (August 2018), 5 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

In this project, we present an implementation of the Ball Pivoting Algorithm (BPA) [Bernardini et al. 1999] for surface reconstruction using point cloud data from The Stanford 3D Scanning Repository. The algorithm takes a list of surface-sample data points as input. It relies on efficient spatial queries, seed selection, ball pivoting, join and glue operations, out-of-core extensions, and multiple passes for effective operation. We first visualize the dataset with the "pptk" library and apply transformations to obtain correctly located point cloud data. The BPA is then employed as an incremental surface reconstruction method that uses an advancing-front paradigm.

We implement and utilize the Octree data structure to optimize spatial queries, which is widely used in computer graphics and computational geometry applications. The BPA consists of two main steps: finding a seed triangle and expanding the triangulation. Finally, we compare our implementation with Open3D BPA and discuss potential improvements to our implementation, including dynamic radius adjustment to address holes in the reconstructed mesh and parallelization techniques to enhance performance. With these refinements, the authors aim to achieve results comparable to the Open3D BPA implementation with these refinements.

Authors' addresses: I-Lun Tsai, University of California, Berkeley, Berkeley, USA, iluntsai99@berkeley.edu; Kevin Xiong, University of California, Berkeley, Berkeley, USA, kevinxiong@berkeley.edu; Sean Wang, University of California, Berkeley, Berkeley, USA, sean.wang@berkeley.edu; Shu-Ping Chen, University of California, Berkeley, Berkeley, USA, shuping_chen@berkeley.edu.


0730-0301/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

## 2 METHOD

### 2.1 Dataset

We use The Stanford 3D Scanning Repository as our dataset to test our results. To begin with, we read point cloud data from a single PLY file format and displayed it using the "pptk" (Point Processing Toolkit) library. However, if we load all the point cloud data and visualize them directly, all the points will blend together in Figure 1. In order to get correctly located point cloud data, we have to load the quaternions and translations to get the transformation matrix from camera view to world view. After transforming the coordinates correctly, the point cloud data shown in Figure 2 looks reasonable.

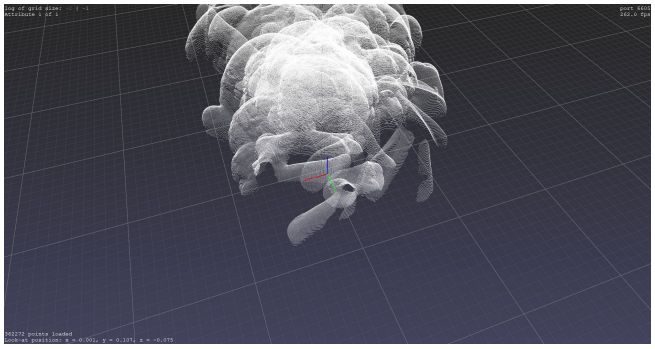

Fig. 1. Merge all the points without transforming their coordinates.

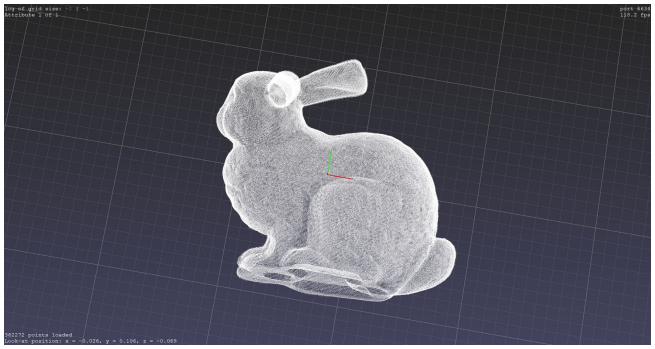

Fig. 2. Merge all the points by transforming their coordinates.

### 2.2 The Ball-Pivoting Algorithm

The Ball Pivoting Algorithm (BPA) is a surface reconstruction method that uses an advancing-front paradigm to build an interpolating triangulation incrementally. The BPA works by finding a seed triangle

and adding one triangle at a time by performing the ball pivoting operation. The algorithm takes as input a list of surface-sample data points, each associated with a normal vector and a ball radius parameter.

The front of the algorithm is represented as a collection of linked lists of edges, initially composed of a single loop containing three edges defined by the first seed triangle. Each edge in the front is represented by its two endpoints, the opposite vertex, the center of the ball touching all three points, and links to the previous and next edge along the same loop of the front. An edge can be active, boundary, or frozen.

The basic BPA algorithm requires efficient spatial queries, seed selection, ball pivoting, join and glue operations, out-of-core extensions, and multiple passes to work effectively. The algorithm is linear in the number of data points and uses linear storage, assuming the data density is bounded.

Spatial queries are implemented using a regular grid of cubic cells or voxels, in our project. we utilize the Octree data structure to save time and memory, more details will be introduced in the next section. Seed selection is essential for reconstructing the entire manifold, and an efficient seed-searching strategy is needed in the presence of noisy, incomplete data. Ball pivoting operations generate triangles while adding and removing edges from the front loops. The join and glue operations handle cases where a newly encountered data point is touched, or a previously used one is encountered. The multiple-pass technique deals with unevenly sampled surfaces by running the algorithm multiple times with increasing ball radii. The out-of-core extensions enable the algorithm to triangulate large datasets with minimal memory usage.

Overall, the Ball Pivoting Algorithm is an effective surface reconstruction method suited for scanned data collected by equipment with a known sample spacing.

Fig. 3. Mesh representations generated from Open3D Library

## 2.3 Implementation detail: Octree (Spatial Queries)

An octree is a hierarchical data structure used in computer graphics, computational geometry, and spatial indexing to represent and organize 3D space efficiently. It extends the binary tree (1D) and quadtree (2D) structures to three dimensions.

In an octree, each node represents a cubic volume of the 3D space. The root node represents the entire space, and its children represent

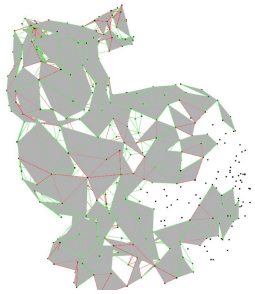

Fig. 4. Bugged ball pivoting implementation

eight equal-sized sub-cubes obtained by dividing the parent cube into eight smaller cubes. This subdivision process continues recursively until a predefined depth is reached or a specific condition is met, such as a maximum number of objects within a node.

Octrees are commonly used in various applications, including 3D computer graphics, collision detection, image processing, point cloud processing, and spatial data indexing. They are particularly useful for tasks that involve searching or traversing 3D spaces efficiently, as they allow for quick queries and updates based on spatial proximity.

To better understand the implementation of Octree, let's break down the process into smaller, more manageable concepts.

(1) Octree: An Octree is a data structure used for spatial partitioning, where each internal node has up to eight children. It helps organize 3D data into smaller sections for efficient queries and processing.
(2) Locational Codes: These codes speed up neighbor searches in the Octree. A locational code represents the position of a point or cell within the Octree in binary notation, making it easier to navigate through the structure.
(3) Octree Construction: The Octree is built using the input points and a specified depth. Each point is assigned a locational code, and the Octree is created by recursively dividing the space and assigning the points to the appropriate cells.
(4) Neighborhood Queries: A shown in 1, This is the process of finding all the neighbors of a given point within a specified radius. The algorithm uses the Octree and locational codes to efficiently search for neighbors without checking all points in the dataset.

Let's take a closer look at how the algorithm works, step by step:

(1) Determine the Octree depth corresponding to the input radius.
(2) Find the cell at that depth level that contains the query point.
(3) Identify the neighboring cells at that depth level that intersect with the query sphere (centered at the query point with a radius equal to the input radius).
(4) Iterate through the points in those neighboring cells and add any point within the specified radius to the set of neighbors.

Using the Octree data structure and locational codes, the algorithm can efficiently find neighbors of a given point without needing

to check all the data points in the dataset. Octree greatly improves the speed and performance of neighborhood queries, making it a popular choice for many computer graphics and computational geometry applications.

**Algorithm 1:** $N_r(p) = getNeightbors(p, P, r)$. Finding the neighbors with fixed radius of a given point. $P$: point cloud, $N_r(p)$ the range neighborhood of radius $r$ centered at $p$, i.e. $N_r(p) = \{q \in P | \|q - p\| \leq r\}$

**Input :** An octree containing a point cloud $P$, a query point $p$, a radius $r$
**Output :** A set of neighbors $N_r(p)$

$l \leftarrow$ depth corresponding to radius $r$
$C_l \leftarrow$ cell containing $p$ at level $l$
$Cells \leftarrow$ cells of level $l$ adjacent to $C_l$ and intersecting $B(p, r)$
**for** $C \in Cells$ **do**
  **for** $q \in C$ **do**
    **if** $\|p - q\| \leq r$ **then**
      $q$ is added to $N_r(p)$
    **end**
  **end**
**end**

## 2.4 Implementation detail: BPA algorithm

The Ball Pivoting Algorithm (BPA) consists of two main steps: finding a seed triangle and expanding the triangulation. Finding a seed triangle is the initial step done using a heuristic to choose a "good" starting triangle. A good triangle means the 3D ball with a given radius $\gamma$ passes through three points in the point cloud and does not contain other points. The expanding step is adding new triangles by pivoting the ball around the edges in previous triangles until no more edge remains. As shown in Algorithm 2, the loop step will not stop until no more seed triangles can be found. In our implementation, we follow the detail discussed in [Digne 2014]

**Algorithm 2:** Ball Pivoting Reconstruction Overview

function BPA $(P, \gamma)$;
**Input :** Point cloud $P$, a ball radius $\gamma$
**Output :** A surface mesh $S$
**while** *A new seed triangle can be found* **do**
  $T = FindSeedTriangle(P, \gamma)$
  $S+ = ExpandTriangle(T, P, \gamma)$
**end**

(1) Finding a Seed Triangle
This starts by selecting a point and searching for pairs of neighboring points that can form a triangle with the initial point. The triangle should have a circumsphere of radius r (the ball) with an empty interior. In practice, multiple triangles may fit this description. This step is illustrated in Algorithm 3, the loop step will not stop until no more seed triangles can be found.

(2) Expanding the Triangulation
After finding a seed triangle, the algorithm expands the triangulation by iteratively rotating the ball around the triangle's edges. The ball is rotated until it meets another point, v, and checks if it remains empty. If the ball is empty, a new facet (triangle) is created using the edge and point v. Depending on the situation. The algorithm can perform the following operations:

- Expansion case: v is an orphan vertex (not part of any triangle). A new facet is created, and the front edges are updated accordingly.
- Gluing case: v is not an orphan vertex but is not linked to the end vertices of the edge. A new facet is created, and the front edges are updated accordingly.
- Hole filling case: v is already linked to the end vertices of the edge. A new facet is created, and the front edges are updated accordingly.
- Ear filling case: v is linked to only one of the edge's end vertices. A new facet is created, and the front edges are updated accordingly.

The algorithm ensures manifold vertices and edges by checking certain conditions during triangulation. The process continues until no more edges can be expanded. As shown in Algorithm 4.

**Algorithm 3:** Finding a seed triangle

FindSeedTriangle $(P, \gamma)$;
**Input :** Point cloud $P$, a ball radius $\gamma$
**Output :** A seed triangle $T$
**for** $p \in P$ **do**
  Look for all points in the neighborhood $N_2r(p)$ of point $p$ sorted by increasing distance
  **for** $(q, s) \in N_2r(p)$ **do**
    **if** $p, q, s$ *are in empty ball configuration* **then**
      return seed triangle$(p, q, s)$
    **end**
  **end**
**end**

# 3 RESULTS

In this section, we present the comparison of our Ball Pivoting Algorithm (BPA) implementation with the Open3D BPA implementation. Our results show that our implementation effectively reconstructs the surface mesh from point clouds, but it is less accurate and slower than the Open3D implementation.

(1) Surface Mesh Reconstruction
Upon testing our BPA implementation on various point cloud datasets, we noticed that the generated surface meshes still have holes as you can see in 6 and 7, indicating some bugs in our algorithm. These holes can be attributed to the tradeoff of choosing smaller balls to have more details. Although our implementation does produce a decent approximation of the

**Algorithm 4:** Expanding the triangle

```
Expand Triangle (T, P, γ);
Input  : A seed triangle T, Point cloud P, a ball radius γ
Output : A surface mesh S
Add the three edges of the seed triangle T to set E
while E is not empty do
    e = E.pop()
    v = newCandidate(e, P, γ)
    if v == NULL then
        Tag e as boundary edge
        Continue
    end
    Add triangle based on (e, v) and add it to S
    add (e.source, v) to E
    add (e.target, v) to E
end
```

original surface, it is evident that the Open3D implementation (shown in 5) provides a more accurate and complete reconstruction.

(2) Performance Comparison
Regarding computation time, we observed that the Open3D BPA implementation is faster than ours. This difference in performance can be due to the more optimized data structures and parallelism algorithms employed by Open3D, which allows for faster processing of the point cloud data.

Fig. 5. Mesh reconstructed by using Open3D API

## 4 DISCUSSION

In this section, we discuss potential improvements to our Ball Pivoting Algorithm (BPA) implementation that could address the issues of holes in the reconstructed surface mesh and improve the overall performance.

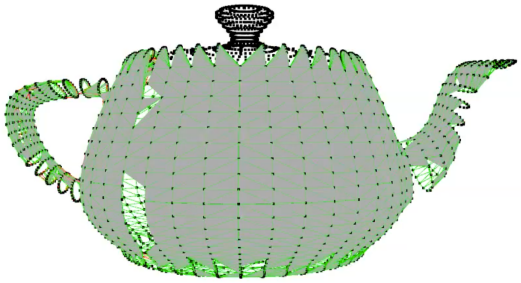

Fig. 6. Our implementation of BPA that reconstructs teapot

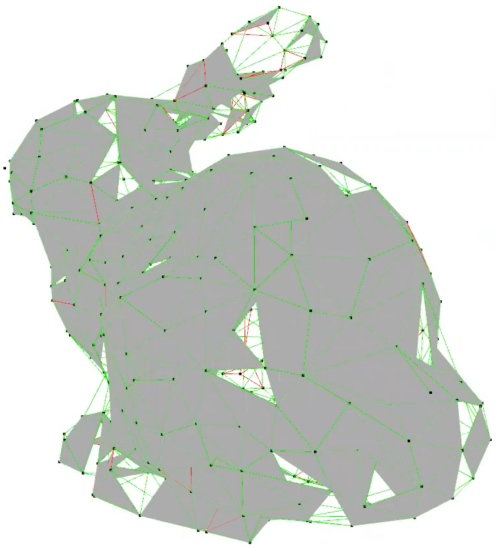

Fig. 7. Our implementation of BPA that reconstructs bunny

(1) Dynamic Radius Iteration
One possible approach to addressing the holes in the reconstructed mesh is to use an iterative, dynamic radius adjustment. Currently, our algorithm uses a fixed radius for the ball pivoting process. However, this may not be suitable for handling point clouds with varying point densities, which can result in holes in the reconstructed mesh.
By implementing a dynamic radius adjustment, our algorithm could adapt the ball radius based on the local point density in the neighborhood of each seed triangle. This would allow the algorithm to better handle regions of varying point density, potentially reducing the occurrence of holes in the reconstructed mesh. Further experimentation and validation are needed to determine the optimal strategy for dynamically adjusting the radius during the reconstruction process.

(2) Parallelization Techniques
To improve the performance of our BPA implementation,

we propose incorporating parallelization techniques, such as calculating the neighbors of multiple seed triangles simultaneously. By leveraging the parallel processing capabilities of modern hardware, our algorithm could significantly reduce the computation time required for mesh reconstruction.
One possible approach to parallelize our algorithm is to employ a data-parallel strategy, where different portions of the point cloud data are processed concurrently. Alternatively, we could explore task-parallel strategies that distribute the workload of calculating neighbors and updating the mesh across multiple threads or processing units.

Our end goal is to refine our BPA implementation to match the quality and performance of the Open3D BPA implementation. By addressing the issues of holes in the reconstructed mesh with dynamic radius adjustment and improving the performance with parallelization techniques, we believe our algorithm has the potential to achieve comparable results to Open3D.

## 5 GITHUB REPO

https://github.com/scarletclaw/Ball-Pivoting-Algorithm

## REFERENCES

F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. 1999. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics* 5, 4 (1999), 349–359. https://doi.org/10.1109/2945.817351

Julie Digne. 2014. An Analysis and Implementation of a Parallel Ball Pivoting Algorithm. *Image Processing On Line* 4 (2014), 149–168. https://doi.org/10.5201/ipol.2014.81.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009