Chapter 6: Polygons, Polyhedra, and Topological Constructs

Probabilistic Minds Consortium (voids.blog)

2024/2025

1 Introduction

Thus far, we have seen how finite, grains-coded increments, such as

$$\frac{k}{N(a)},$$

can represent distances, capacities, and arithmetic. In this chapter, we apply these ideas to geometric shapes—specifically, to polygons, loops, and higher-dimensional meshes—using discrete adjacency rather than continuous geometry.

Why do this?
Real-number geometry can introduce floating-point precision errors and hidden infinities. In contrast, a grains-coded adjacency approach keeps every relationship exact and finite. Geometric structures such as polygons and polyhedra built in this manner are composed of precisely defined, discrete steps. This is crucial for both theoretical rigor and practical applications, where a clear, unambiguous definition of "inside" and "outside" is essential.

2 Polygonal Loops in a Discrete Space

2.1 Polygons as Loops of Adjacent Points

A polygon can be viewed as a closed sequence of grains-coded points, where each consecutive pair is considered "adjacent." In grains-coded geometry, adjacency can be defined in one of two ways:

1. **Distance-Based:** Two points P_i and P_{i+1} are adjacent if the grains-coded distance (using, for example, a Manhattan or L1 metric) satisfies

$$\mathrm{Dist}(P_i, P_{i+1}) = \frac{1}{N(a)}.$$

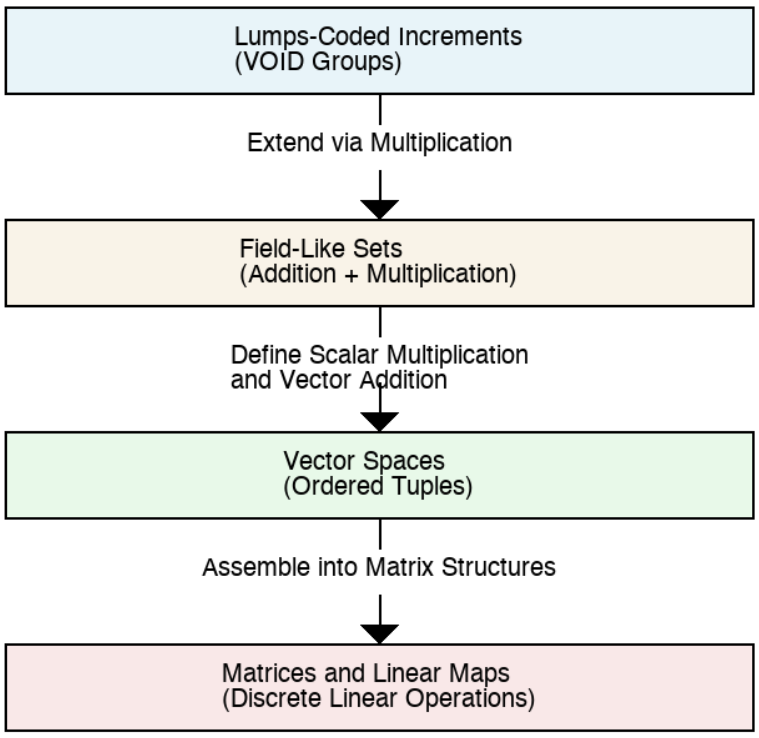
2. **Graph-Based:** Adjacency is defined by an explicit edge connecting the two points; here the system records a list of edges, each connecting a pair (P_i, P_{i+1}).

We say that the loop is polygonal if:

- Every consecutive pair P_i, P_{i+1} (for $i = 1, 2, \ldots, n-1$) forms an edge, and
- The wraparound edge (P_n, P_1) is also present.

Formally, if we define a predicate $\mathrm{adj}(P_i, P_j)$ to mean "P_i and P_j are neighbors," then the loop condition is:

$$\mathrm{Loop}(P_1, \ldots, P_n) \quad \Longleftrightarrow \quad \Big(\forall\, i = 1, \ldots, n-1 :\ \mathrm{adj}(P_i, P_{i+1})\Big) \wedge \mathrm{adj}(P_n, P_1).$$



Additional Explanation:
Using the L1 metric, a check such as `grains_l1_distance(P_i, P_j) == 1/N(a)` confirms that two points are exactly one minimal step apart. This guarantees that the polygon is built from the smallest, exact increments available.

2.2 Inside/Outside Partition

Removing the edges of a closed loop from your adjacency graph partitions the graph into two distinct regions: an inside and an outside.

- Let G be a grains-coded adjacency graph of points.
- If E is the set of edges forming the loop, remove these edges from G.
- Choose a known "exterior" node v. Then, any node reachable from v in the modified graph is considered outside; nodes that are disconnected are considered inside.

This discrete method of region partitioning resolves the common ambiguity in computational geometry regarding inside/outside relationships by relying solely on connectivity—without any floating-point computations or ambiguous boundary cases.

Implementation Sketch:

```
def dfs_outside(ref_node, loop_edges):
    visited = set()
    stack = [ref_node]
    visited.add(ref_node)
    while stack:
        curr = stack.pop()
        for neigh in curr.neighbors:
            # Skip edges that belong to the loop
            if (curr, neigh) not in loop_edges and (neigh, curr) not in loop_edges:
                if neigh not in visited:
                    visited.add(neigh)
                    stack.append(neigh)
    return visited
```

Any node not in the returned visited set is considered "inside."

3 Higher-Dimensional Complexes

3.1 Beyond 2D: 3D Meshes or nD Graphs

The same grains-coded adjacency logic that constructs 2D polygons generalizes naturally to three-dimensional meshes or even higher-dimensional graphs:

- In a 3D grains-coded mesh, each node is represented with grains-coded coordinates. Adjacency defines edges or faces.
- A "face" might be a closed 2D loop in the 3D space, and a "cell" might be an enclosed volume.

The beauty of this approach is its scalability—adding another dimension simply involves adding another coordinate, while the principles of discrete increments and adjacency remain unchanged. This property is particularly valuable in fields like computer graphics, simulation, and physical modeling.

3.2 Example: 2D Mesh with Neighbors

Consider a script (e.g., $2\mathrm{d}_3 d_m esh.py) that builds a grains-coded 2D grid$:

```
class MeshNode:
    def __init__(self, x_grain, y_grain):
        self.x = x_grain
        self.y = y_grain
        self.neighbors = []

def build_2d_mesh(nx, ny):
    nodes = []
    # Create nodes with grains-coded coordinates
    for i in range(nx):
        row_nodes = []
        for j in range(ny):
            node = MeshNode(grain(i), grain(j))
            row_nodes.append(node)
        nodes.append(row_nodes)
    # Link neighbors (4-way adjacency)
    for i in range(nx):
        for j in range(ny):
            node = nodes[i][j]
            for (di,dj) in [(1,0),(-1,0),(0,1),(0,-1)]:
                ni, nj = i+di, j+dj
                if 0 <= ni < nx and 0 <= nj < ny:
```

```
                node.neighbors.append(nodes[ni][nj])
    return nodes
```

This function creates a grid where each node has grains-coded coordinates and is linked to its adjacent neighbors. Extending this to 3D (or higher dimensions) simply requires adding more loops and additional coordinate handling.

4 Practical Workflow for Discrete Geometry

1. **Generate a Mesh or Graph:** Create grains-coded points (as in a 2D or 3D grid) and define adjacency using either distance-based criteria or explicit neighbor lists.
2. **Identify Polygons:** Detect closed loops within the adjacency graph. Store these loops as sequences of grains-coded points.
3. **Inside/Outside Partitioning:** Remove the loop's edges, then perform a breadth-first search (BFS) or depth-first search (DFS) from a known exterior node. Points not reached are deemed "inside."
4. **Extend to Higher Dimensions:** Use the same principles for 3D or nD spaces. Define edges, faces, and volumes using grains-coded increments and explicit adjacency lists.

5 Why This Matters

- **Discrete Shapes:** In many fields—such as simulation, gaming, or GIS—shapes defined using grains-coded adjacency eliminate floating-point errors, improving the reliability of collision detection, region partitioning, and pathfinding.
- **Exact Partitioning:** Inside/outside decisions become unambiguous: once the loop edges are removed, connectivity alone determines region membership.
- **Unified Finite Model:** Just as with grains-coded probabilities and arithmetic, discrete geometry remains entirely finite. All geometric constructs are defined as a finite set of increments $\frac{k}{N(a)}$.
- **Scalability:** When finer resolution is needed, capacity can be refined—but always in a finite, controlled manner. There is no "infinite plane" or continuous manifold; instead, you get a denser finite grid.

This approach replaces continuous geometry with an exact, discrete model that eliminates common errors and ambiguities. Every geometric construction is built from countable, exact steps—ideal for systems where precision and stability are critical.

6 Applications and Conclusion

Finite, grains-coded geometry has significant applications in:

- **Computer Graphics:** Improving rendering accuracy and collision detection through exact geometric calculations.
- **Scientific Simulation:** Enhancing stability in simulations by avoiding floating-point approximation errors.
- **Geographic Information Systems (GIS):** Providing unambiguous inside/outside region determinations.
- **Computer-Aided Design (CAD):** Enabling precise geometric modeling with exact arithmetic.

Conclusion: Polygons, loops, polyhedra, and topological partitions can be represented entirely in a grains-coded system. By relying on discrete, exact increments, we eliminate the floating-point drift and numerical instabilities common in traditional continuous geometry. From 2D polygons to 3D meshes, every construct is computed using well-defined, finite operations. This method not only maintains mathematical rigor but also opens up new possibilities for applications in computer graphics, simulation, GIS, and CAD—providing robust, resource-aware solutions for real-world problems.