

# Road Parceling System

Design Document

Dane Sabo  
(with Claude Code)

Started 2026-04-25

## Abstract

This is the *contract* for the road parceling system — the foundational geometric layer of a city simulation game where parcels are arbitrary polygons drawn outward from road frontage rather than cells in a fixed grid. It defines invariants, algorithms, the public API, the testing strategy, and the milestone roadmap; it includes a first-principles walkthrough of how the implementation works (section 17) so a reader can understand the geometry without spelunking through source. The companion `journal.tex` is the running log of work, decisions, and deviations.

## Contents

<b>1</b>	<b>Project Context and Motivation</b>	<b>4</b>
1.1	Why Build This . . . . .	4
1.2	Scope of This Document . . . . .	4
1.3	Audience . . . . .	4
<b>2</b>	<b>Core Invariants</b>	<b>4</b>
<b>3</b>	<b>Geometric Foundations</b>	<b>5</b>
3.1	Coordinate System and Numerical Type . . . . .	5
3.2	Road Network as a Planar Graph . . . . .	6
3.3	Block Extraction . . . . .	6
3.4	Inward Offsetting . . . . .	6
<b>4</b>	<b>Subdivision Algorithm</b>	<b>6</b>
4.1	Frontage-First Subdivision . . . . .	6
4.2	Edge Classification . . . . .	7
4.3	Regularization Pass . . . . .	8
<b>5</b>	<b>Road Edit Handling</b>	<b>8</b>
5.1	Edit Types . . . . .	8
5.2	Deformation Pipeline . . . . .	8
5.3	Regeneration Thresholds . . . . .	9
5.4	Building Footprint Preservation . . . . .	9
<b>6</b>	<b>Degenerate Cases</b>	<b>9</b>

<b>7</b>	<b>Crate Architecture</b>	<b>11</b>
7.1	Module Layout . . . . .	11
7.2	Public API Surface . . . . .	11
7.3	Error Types . . . . .	12
7.4	Dependencies . . . . .	12
<b>8</b>	<b>Idiomatic Rust Requirements</b>	<b>12</b>
<b>9</b>	<b>Testing Strategy</b>	<b>13</b>
9.1	Three Layers . . . . .	13
9.2	Snapshot Testing . . . . .	13
9.3	Coverage Requirement . . . . .	14
<b>10</b>	<b>Visualization and Figures</b>	<b>14</b>
10.1	Required Figures . . . . .	14
10.2	Color Conventions . . . . .	14
<b>11</b>	<b>Performance Targets</b>	<b>14</b>
<b>12</b>	<b>Out of Scope</b>	<b>15</b>
<b>13</b>	<b>Claude Code Contract</b>	<b>15</b>
<b>14</b>	<b>Roadmap</b>	<b>16</b>
<b>15</b>	<b>Open Questions</b>	<b>17</b>
<b>16</b>	<b>Design Decisions Index</b>	<b>18</b>
<b>17</b>	<b>System Walkthrough</b>	<b>21</b>
17.1	Pipeline at 30,000 ft . . . . .	21
17.2	Mathematical primitives . . . . .	21
17.2.1	Rotations . . . . .	21
17.2.2	Polygon orientation: the shoelace formula . . . . .	22
17.2.3	Segment intersection . . . . .	23
17.2.4	Polygon-vs-half-plane clipping (Sutherland–Hodgman) . . . . .	23
17.3	The road graph as a DCEL . . . . .	24
17.3.1	Why DCEL . . . . .	24
17.3.2	Half-edge structure . . . . .	24
17.3.3	Constructing the DCEL: the next-pointer rule . . . . .	24
17.3.4	Face extraction . . . . .	25
17.4	Block extraction . . . . .	26
17.5	Frontage-first subdivision . . . . .	26
17.5.1	Interior angles and corner classification . . . . .	26
17.5.2	Per-edge depth cap (ray-cast) . . . . .	27
17.5.3	Corner parcel construction . . . . .	27
17.5.4	The frontage walk . . . . .	28
17.5.5	Block-boundary defense clip . . . . .	28
17.6	Shared-vertex registry . . . . .	28

---

17.6.1	Data structure . . . . .	29
17.6.2	Snap-on-insert . . . . .	29
17.6.3	Write-through propagation . . . . .	30
17.7	Edit pipeline . . . . .	30
17.7.1	Move-node case . . . . .	30
17.8	End-to-end trace: subdivide_all . . . . .	31
17.9	End-to-end trace: apply_road_edit(MoveNode) . . . . .	32
<b>A</b>	<b>Notation Reference</b>	<b>33</b>

# 1 Project Context and Motivation

## 1.1 Why Build This

Modern city simulation games suffer from two architectural choices that compound poorly. First, they tend toward rigid grid-based or cell-based zoning, which produces visually uniform cities that diverge from how real urban form develops parcel by parcel along road frontage. Second, they over-rely on bottom-up agent simulation: every citizen is an autonomous decision-maker rerolling actions on every tick. This scales badly — *Cities: Skylines II* is the canonical example of a game shipped with simulation costs that do not survive contact with a real player’s city.

This project addresses the first problem directly: a parcel-based zoning model where parcels are arbitrary polygons drawn outward from roads, with user-configurable depth and frontage. The simulation architecture (which addresses the second problem via aggregate / hazard-rate modeling rather than per-agent rerolls) is documented separately and consumes the parcel system as a foundational layer.

## 1.2 Scope of This Document

This is the design document for the road parceling system: a pure-logic Rust crate with no rendering engine, no game loop, and no simulation behavior. Downstream systems — buildings, zoning types, population dynamics, transit — consume this crate’s API but are out of scope here. Section 14 sketches what comes after the parceling crate is done.

## 1.3 Audience

The primary audience is future-me. The secondary audience is an autonomous coding agent (Claude Code) that implements the spec laid out in section 13. Sections marked as *Claude Code Contract* are written to be acted upon directly. Section 17 is written for a graduate-level engineering reader who knows linear algebra and basic algorithmics but is not a computational-geometry specialist; it derives the geometric machinery from first principles.

# 2 Core Invariants

These are the load-bearing properties of the system. Every public function must preserve them, and every test suite must verify them after every operation. They are stated here once and referenced by number throughout the rest of the document.

### I1: Polygon validity

Every parcel is a simple polygon: no self-intersections, no holes, vertices ordered counter-clockwise. No edge has length less than  $\varepsilon_{\text{geom}}$ . No three consecutive vertices are collinear within  $\varepsilon_{\text{angle}}$ .

### I2: Single frontage

Each parcel has exactly one edge classified as `EdgeKind::Frontage`, lying coincident (within  $\varepsilon_{\text{geom}}$ ) with a road segment in the network.

**I3: Non-overlap**

For any two parcels  $P_i, P_j$  within the same block, the area of their interior intersection is zero within  $\varepsilon_{\text{area}}$ .

**I4: Edit persistence**

When a road edit modifies a segment  $s$ , parcels with frontage on  $s$  recompute only their frontage edge. Non-frontage edges are preserved unless explicit geometric thresholds (section 5) force regeneration.

**I5: No degenerate output**

The public API never returns parcels violating I1–I3. Inputs that would produce such parcels are either gracefully merged with neighbors, regularized, or rejected with a typed error. The library never panics on invalid input.

**I6: Edit determinism**

Applying the same `RoadEdit` to the same `ParcelSet` twice produces identical output, byte-for-byte (modulo opaque IDs).

**I7: Edit reversibility**

Applying an edit and then its inverse restores the original parcel set within  $\varepsilon_{\text{geom}}$  for all preserved parcels. Condemned parcels are not restored; this is a known asymmetry and acceptable. (See journal entry on minimum-change deformation: a centroid-bounded reading of I7 is what the implementation actually delivers.)

**I8: Shared-vertex consistency**

Two parcels whose polygons would coincide on a vertex hold the same `VertexId` for that point. `ParcelSet::move_vertex` writes through to every referrer simultaneously, so adjacent parcels' shared boundaries cannot drift apart under repeated edits.

The numerical tolerances are crate-wide constants:

$$\begin{aligned}\varepsilon_{\text{geom}} &= 10^{-6} \text{ m} \\ \varepsilon_{\text{area}} &= 10^{-9} \text{ m}^2 \\ \varepsilon_{\text{angle}} &= 10^{-4} \text{ rad}\end{aligned}$$

## 3 Geometric Foundations

### 3.1 Coordinate System and Numerical Type

All geometry is 2D, in a flat Euclidean plane with units of meters. Coordinates use `glam::DVec2` (double-precision) throughout. Single-precision `Vec2` is rejected because parcel offset operations on long road segments accumulate error rapidly at `f32` resolution; at city scales of  $10^4$  m, an `f32`

mantissa gives roughly  $10^{-3}$  m precision, which is insufficient for the cleanup passes described in section 4.3.

### 3.2 Road Network as a Planar Graph

The road network is represented as a planar graph  $G = (V, E)$  where vertices are intersections and edges are road segments. We use a half-edge / DCEL (doubly-connected edge list) representation because it provides  $O(1)$  access to:

- the next edge around a face (block boundary traversal),
- the twin edge across a road (parcels on the other side),
- the edges incident to a vertex (intersection topology).

Faces of the planar graph correspond to blocks. The unbounded exterior face is excluded from subdivision. The DCEL construction algorithm and the next-pointer rule are derived from first principles in section 17.3.

### 3.3 Block Extraction

A block is a closed face of the planar graph other than the unbounded exterior. Extraction proceeds by:

1. Identifying all faces of the DCEL via half-edge traversal.
2. Computing the signed area of each face; the unique face with negative area (under CCW convention) is the exterior.
3. Returning the remaining faces as block boundaries.

### 3.4 Inward Offsetting

Given a block boundary  $B$  as a CCW polygon and a setback distance  $d_s$ , the developable polygon  $B'$  is the inward offset of  $B$  by  $d_s$ . For convex inputs this is exact; for concave inputs it is the intersection of inward half-planes (a conservative approximation that may shave concave outer-corner detail but never produces an invalid polygon).

The offset can fail in two ways:

1. For very narrow blocks,  $B' = \emptyset$ . The block is then unbuildable and produces zero parcels.
2. For non-convex blocks, the offset may produce multiple disjoint polygons. Each component is subdivided independently.

## 4 Subdivision Algorithm

### 4.1 Frontage-First Subdivision

The primary algorithm subdivides a block by walking along its road-facing boundary in increments of approximately the target frontage width, extruding perpendicular into the block interior. This produces parcels that face their road, which is the desired aesthetic and the realistic outcome.

**Inputs.** A block boundary  $B$  (CCW), the developable polygon  $B' = \text{offset}_{-d_s}(B)$ , and parameters:

$$\theta = (w_f, \sigma_f, d_p, \sigma_d, \rho, w_{\min}, A_{\min}, \text{seed})$$

where  $w_f$  is target frontage width,  $\sigma_f$  is frontage variance,  $d_p$  is target depth,  $\sigma_d$  is depth variance,  $\rho \in [0, 1]$  is the regularity slider,  $w_{\min}$  is minimum frontage, and  $A_{\min}$  is minimum area.

### Procedure (high level).

1. Identify “real corners” of the block boundary — vertices where the underlying road graph node has degree  $\geq 3$ , or vertices where the block boundary turns sharply enough to warrant a corner parcel rather than a continuous frontage walk. Acute corners (interior  $< 60^\circ$ ) are flagged separately.
2. At each real corner, build a corner parcel: an axis-aligned (or skewed for non-90° corners) rectangle of dimensions  $R$  by depth, anchored at the corner vertex, with frontage on the longer of the two adjacent block edges.
3. For each block boundary edge, walk the segment between the corner footprints (if any) at arc-length intervals  $w_i = w_f + \sigma_f \cdot \xi_i$  where  $\xi_i \sim \text{Uniform}(-1, 1)$  from a deterministic per-road RNG.
4. At each split point, extrude perpendicular into the block by a depth bounded by a per-edge ray-cast cap (half the perpendicular distance from the edge midpoint to the nearest other block edge).
5. Form quadrilateral parcels from consecutive split points and their extrusions.
6. Clip each parcel polygon against the block boundary’s inward half-planes to ensure no parcel can extend past the block.
7. Reject parcels with frontage  $< w_{\min}$  or area  $< A_{\min}$ .

The algorithmic details, including the corner classification math, the ray-cast depth cap formulation, and the bisector-clip fallback for acute corners, are derived in section 17.5.

## 4.2 Edge Classification

After subdivision, each parcel edge is classified:

Kind	Definition	Color (figures)
Frontage	Lies within $\varepsilon_{\text{geom}}$ of a road segment	Blue
Side	Adjacent to the frontage edge in the polygon ring	Gray
Back	All other edges	Light gray, dashed

Table 1: Edge classification scheme.

For non-quadrilateral parcels (e.g. pie slices in cul-de-sacs, sliver-merged parcels with extra vertices), the back classification absorbs all non-frontage, non-side edges.

### 4.3 Regularization Pass

When  $\rho > 0$ , a regularization pass runs after subdivision. For each parcel:

1. Compute the OBB (oriented bounding box) of the parcel, oriented to the frontage edge.
2. Linearly interpolate side-edge vertices toward their OBB-snapped positions with weight  $\rho$ .
3. Validate the result against I1; if validation fails, revert.

At  $\rho = 1$ , parcels are forced to perfect rectangles aligned to their road. At  $\rho = 0$ , parcels carry whatever shape the raw subdivision produced. Intermediate values give partial cleanup, useful for cities where some neighborhoods should look planned and others organic.

## 5 Road Edit Handling

The most distinctive feature of this system is parcel persistence under road edits. Conventional city builders nuke and re-create parcels (and their buildings) when roads change. Here, parcels survive whenever geometrically reasonable, and shared boundary vertices update through the registry so adjacent parcels stay in lockstep.

### 5.1 Edit Types

```

1 pub enum RoadEdit {
2     MoveNode { node: NodeId, to: DVec2 },
3     SplitSegment { road: RoadId, at: DVec2 },
4     DeleteSegment { road: RoadId },
5     InsertSegment { from: NodeId, to: NodeId },
6 }

```

Listing 1: Road edit enum.

### 5.2 Deformation Pipeline

When `apply_road_edit` is invoked:

1. **Snapshot the graph** pre-edit so deform can re-project parcel frontages from the old geometry onto the new.
2. **Apply the topology mutation** and rebuild the DCEL.
3. **Identify affected parcels:** those with frontage on a modified road.
4. **Propose** a vertex-move set per affected parcel (no in-place mutation yet). Each parcel is categorized:
  - *Untouched*: the road's *line* didn't change (only its endpoints shifted along it) and the parcel's frontage is still inside the new segment. Skip.
  - *Deformed*: project the frontage endpoints onto the new road via the original parameter mapping; validate against rotation/area thresholds.
  - *Condemned*: validation fails (frontage too short, area too small, no longer on road).
  - *Regenerate*: rotation threshold exceeded; the affected block is re-subdivided.



5. **Apply** all proposed vertex moves through `ParcelSet::move_vertex`, which writes through the shared-vertex registry. Adjacent parcels' shared boundaries move atomically (I8).
6. **Building eviction**: for each surviving parcel with an attached building, call `BuildingFitCheck::fits_in`; on `false` the building is evicted but the parcel survives.
7. **Drop** condemned parcels.
8. **Regenerate** marked blocks.

The full derivation, including the parameter projection and the line-unchanged check, lives in section 17.7.

### 5.3 Regeneration Thresholds

Deformation triggers regeneration of the affected block when any of the following hold for the deformed parcel:

Condition	Outcome
Frontage length $< w_{\min}$	Condemned
Side edge rotated $> \alpha_{\max}$ from original	Regenerated
Polygon self-intersects	Regenerated
Area $< A_{\min}$	Condemned
Frontage no longer adjacent to any road	Condemned

Table 2: Thresholds that trigger regeneration or condemnation.  $\alpha_{\max}$  defaults to  $30^\circ$ .

### 5.4 Building Footprint Preservation

Parcels carry an opaque `Option<BuildingHandle>`. The crate does not define what a building is, but exposes a hook:

```

1 pub trait BuildingFitCheck {
2     /// Returns true if the building still fits inside the deformed
3     /// parcel.
4     fn fits_in(&self, parcel: &Parcel) -> bool;
5 }
```

Listing 2: Building persistence hook.

If a building's `fits_in` returns false during deformation, the parcel survives but its building is evicted. This is the key behavior that distinguishes the system from CS2's nuke-on-edit approach.

## 6 Degenerate Cases

The correctness of this system is largely defined by how it handles degenerate inputs. Each case below has a named test in the suite.

Test name	Scenario	Expected behavior
<code>acute_intersection_15deg</code>	Two roads meet at $15^\circ$	Sliver merged or rejected; I1–I3 hold
<code>acute_intersection_5deg</code>	Knife-edge angle	No panic; typed error or valid output
<code>colinear_roads</code>	Two segments end-to-end, zero turn	Treated as one continuous frontage
<code>zero_length_segment</code>	Coincident endpoints	Returns <code>InvalidParams</code> or skips
<code>near_duplicate_nodes</code>	Nodes within $\varepsilon$ of each other	Merged or typed error
<code>self_intersecting_graph</code>	Roads cross with no node	Returns <code>NonPlanarGraph</code>
<code>cul_de_sac</code>	Single road into a bulb	Pie-slice parcels tile the bulb
<code>t_intersection</code>	Standard T	All three blocks subdivide
<code>y_intersection</code>	Three roads at $120^\circ$	Corner parcels handled
<code>tiny_block</code>	Perimeter $< 4w_{\min}$	0 or 1 parcel; never invalid
<code>huge_block</code>	1 km $\times$ 1 km block	Sane parcel count; no explosion
<code>curved_road_high_curv</code>	Road radius $< d_p$	No self-intersection
<code>road_edit_micro_move</code>	Move node by 0.01 m	All parcels deformed; none regen
<code>road_edit_large_move</code>	Move node by 50 m	Mix of deformed/regen/condemned
<code>road_edit_inverse_restores</code>	Apply edit then inverse	State matches initial within $\varepsilon$ (centroid-bounded post-D14)
<code>road_delete_condemns</code>	Delete a road segment	All frontage parcels condemned
<code>road_split_preserves</code>	Split segment with new node	Parcels deform; none regenerated
<code>building_footprint_persists</code>	Sub building, deform parcel	Building kept iff <code>fits_in</code> true
<code>degenerate_isolated_node</code>	Graph node with no edges	Skipped; no panic
<code>disconnected_graph</code>	Two components	Each subdivides independently
<code>numerical_precision_stress</code>	Coords near $10^{20}$	I1–I3 still hold

Table 3: Required degenerate-case tests. Each must exist by name and pass.

## 7 Crate Architecture

### 7.1 Module Layout

```

1 road_parceling/
2 |-- Cargo.toml
3 |-- src/
4 |   |-- lib.rs           // public API
5 |   |-- geometry/
6 |   |   |-- polygon.rs   // polygon ops, validation
7 |   |   |-- offset.rs    // road edge offsetting
8 |   |   |-- skeleton.rs  // straight skeleton (M0.5+)
9 |   |-- network/
10 |   |   |-- graph.rs     // DCEL road graph
11 |   |   |-- blocks.rs    // block extraction
12 |   |-- parcel/
13 |   |   |-- subdivide.rs // frontage-first subdivision
14 |   |   |-- classify.rs  // edge classification
15 |   |   |-- deform.rs    // deformation under edits
16 |   |   |-- regularize.rs // OBB snapping
17 |   |-- config.rs        // SubdivisionParams
18 |   |-- error.rs         // typed errors
19 |   |-- viz/svg.rs       // SVG output (feature-gated)
20 |-- tests/
21 |-- examples/
22 |-- benches/
23 |-- figures/             // generated SVG/PDF artifacts

```

Listing 3: Crate structure.

### 7.2 Public API Surface

```

1 pub use config::SubdivisionParams;
2 pub use error::{ParcelError, SubdivisionError};
3 pub use network::{RoadGraph, RoadId, NodeId};
4 pub use parcel::{
5     Parcel, ParcelId, ParcelSet, EdgeKind, VertexId,
6     BuildingFitCheck, BuildingHandle,
7     RoadEdit, EditOutcome,
8     SubdivisionStats,
9 };
10
11 pub fn subdivide_all(
12     graph: &RoadGraph,
13     params: &SubdivisionParams,
14 ) -> Result<ParcelSet, SubdivisionError>;
15
16 pub fn subdivide_all_with_stats(
17     graph: &RoadGraph,
18     params: &SubdivisionParams,
19 ) -> Result<(ParcelSet, SubdivisionStats), SubdivisionError>;
20

```

```

21 pub fn apply_road_edit(
22     parcels: &mut ParcelSet,
23     graph: &mut RoadGraph,
24     edit: RoadEdit,
25     params: &SubdivisionParams,
26 ) -> Result<EditOutcome, ParcelError>;
27
28 pub struct EditOutcome {
29     pub deformed: Vec<ParcelId>,
30     pub regenerated: Vec<ParcelId>,
31     pub condemned: Vec<ParcelId>,
32     pub created: Vec<ParcelId>,
33     pub evicted_buildings: Vec<ParcelId>,
34 }

```

Listing 4: Public API in lib.rs.

### 7.3 Error Types

All fallible operations return `Result`. No panics in library code outside of `debug_assert!`.

```

1 #[derive(Debug, thiserror::Error)]
2 #[non_exhaustive]
3 pub enum SubdivisionError {
4     #[error("road graph is not planar at node {0:?}")]
5     NonPlanarGraph(NodeId),
6     #[error("block boundary is not closed")]
7     OpenBlock,
8     #[error("subdivision parameters invalid: {0}")]
9     InvalidParams(String),
10    #[error("geometric operation failed: {0}")]
11    GeometryFailure(String),
12    #[error("feature not yet implemented: {0}")]
13    Unimplemented(&'static str),
14 }

```

Listing 5: Error enum.

### 7.4 Dependencies

## 8 Idiomatic Rust Requirements

The bar is high; this is a foundational crate that downstream code will depend on for years.

- No `unwrap()` or `expect()` outside tests and examples.
- No `unsafe` without a `// SAFETY: comment`. None is expected.
- Newtype IDs (`ParcelId`, `RoadId`, `NodeId`, `VertexId`); never expose raw indices.
- `#[must_use]` on builders and on `EditOutcome`.
- Iterator-first APIs where allocation is avoidable.

Crate	Version	Purpose
<code>geo</code>	0.28	Polygon primitives, boolean ops (added in M0.5)
<code>glam</code>	0.29	<code>DVec2</code> math
<code>slotmap</code>	1	Stable IDs for graph entities
<code>thiserror</code>	2	Error types
<code>rand</code>	0.8	Deterministic RNG
<code>rand_chacha</code>	0.3	Reproducible RNG backend
<code>svg</code>	0.18	SVG output (feature <code>viz</code> )
<code>serde</code>	1	Serialization (feature <code>serde</code> )
<code>proptest</code>	1	Property-based testing (dev)
<code>insta</code>	1	Snapshot testing (dev)
<code>criterion</code>	0.5	Benchmarking (dev)

Table 4: Dependency manifest.

- Borrowing over cloning; parcels and graphs are large.
- `#[non_exhaustive]` on public enums likely to grow.
- `cargo clippy --all-targets --all-features -- -D warnings clean` (`clippy::all` group; pedantic stays off, see D8).
- `cargo fmt --check clean`.
- `#![deny(missing_docs)]` at crate root; all public items documented.
- Module-level docs at the top of each `mod.rs`, with example snippets where useful.
- Feature flags: `serde`, `viz`.

## 9 Testing Strategy

### 9.1 Three Layers

**Unit tests** live in-module under `#[cfg(test)]`. Every non-trivial geometric helper is tested directly.

**Integration tests** live in `tests/`. They build a road graph, subdivide, and check invariants.

**Property tests** use `proptest`. For each invariant I1–I8, a property test generates random valid road graphs and asserts the invariant. Generators produce graphs with 2–20 nodes, varying segment lengths, intersection angles in  $[30^\circ, 150^\circ]$ , and occasional degeneracies.

### 9.2 Snapshot Testing

Each example scenario in `examples/` renders to SVG and snapshots via `insta`. Visual regressions are caught when the SVG diff changes. Baseline SVGs are committed.

### 9.3 Coverage Requirement

Every named test in table 3 must exist and pass. There is no acceptable substitute. As of M0.4 all 21 are active.

## 10 Visualization and Figures

The crate produces SVG output via the `viz` feature. A dedicated example, `generate_figures`, regenerates every figure referenced in this document.

### 10.1 Required Figures

Filename	Content
<code>fig_01_grid_block.svg</code>	Rectangular block subdivided
<code>fig_02_curved_road.svg</code>	Parcels on a curved frontage
<code>fig_03_cul_de_sac.svg</code>	Pie-slice parcels around a bulb
<code>fig_04_y_intersection.svg</code>	Three-way intersection corner lots
<code>fig_05_acute_corner.svg</code>	Sliver-merge at sharp angle
<code>fig_06a_road_edit_before.svg</code>	Scene before road move
<code>fig_06b_road_edit_after.svg</code>	Same scene after; classes color-coded
<code>fig_07_regularity_slider.svg</code>	$\rho \in \{0.0, 0.5, 1.0\}$ side by side
<code>plot_subdivision_perf.svg</code>	Criterion: parcels/s vs. block count
<code>plot_parcel_area_hist.svg</code>	Histogram, 10k-parcel stress scene

Table 5: Required figure deliverables.

### 10.2 Color Conventions

- Roads: black, 2px stroke
- Frontage edges: blue
- Side edges: gray
- Back edges: light gray, dashed
- Parcel fill: pale yellow, 30% opacity
- Condemned parcels: red fill
- Regenerated parcels: orange fill
- Deformed parcels: green fill
- Created parcels: blue fill

## 11 Performance Targets

The eventual interactive use case (M2 test harness, M3+ game) wants sub-millisecond response on typical edits so placing roads feels instant. Current measurements (M0.4):  $\sim 0.4\text{--}0.7\mu\text{s}$ /parcel on M-series hardware, well under all targets.

Operation	Scale	Target (release)
<code>subdivide_all</code>	100 blocks	< 50 ms
<code>subdivide_all</code>	10 000 blocks	< 5 s
<code>apply_road_edit</code>	10k-parcel graph	< 1 ms per single-segment edit

Table 6: Performance targets. Measured via Criterion.

## 12 Out of Scope

Explicitly *not* part of this milestone, to prevent scope creep:

- Buildings (parcels carry an opaque handle; the crate does not define buildings).
- Zoning types (residential / commercial / industrial). Parcels are typeless.
- Population, agents, simulation tick logic.
- Rendering beyond SVG export.
- Game engine integration (Bevy, Godot).
- 3D / terrain. Everything is 2D in a flat plane.
- Persistence formats beyond optional `serde` derives.
- Multi-threading. Single-threaded for the foundation; APIs designed not to preclude `Send/Sync` later.

## 13 Claude Code Contract

This section is written to be acted on directly by an autonomous coding agent.

### Working Style

1. Work iteratively. Get a single rectangular block working end-to-end with tests and an SVG figure before adding complexity.
2. After each major feature, regenerate figures and verify by inspection.
3. Write tests *before* fixing bugs. Every degenerate case starts as a failing test.
4. When a degenerate case is fundamentally unhandleable (e.g. truly self-intersecting input), the test asserts the correct typed error, not success.
5. Commit frequently with messages naming the feature or invariant addressed.
6. When a design decision has multiple reasonable answers, pick one, document it as a Design Decision in this document (section 16), and move on. Do not block.
7. If the spec is ambiguous or wrong, append a deviation note to the journal (`journal.tex`) listing the deviation. Resolve it later by updating this design document.

### Definition of Done — Milestone 1

The foundational parceling crate is complete (M1.0) when all of the following are simultaneously true:

1. `cargo build --all-features` succeeds with no warnings.
2. `cargo clippy --all-targets --all-features -- -D warnings` passes.
3. `cargo fmt --check` passes.
4. `cargo test --all-features` passes, including every named test in table 3.
5. `cargo doc --all-features --no-deps` produces no warnings.
6. All figures listed in section 10 are generated and committed.
7. Performance targets in section 11 are met on a modern laptop.
8. This design document and the journal both compile with `make`.
9. OBB regularization actually does something at  $\rho > 0$ .

## 14 Roadmap

This section is the project’s current direction. Updated as milestones land.

### M1 — Foundational parceling crate

**Goal.** section 13’s Definition of Done.

**Status (2026-04-26).** ~80% complete. All 21 named tests active and passing as of M0.4; missing items are figures (`fig_03_cul_de_sac`, `fig_05_acute_corner`, `plot_subdivision_perf`, `plot_parcel_area_hist`) and a working OBB regularization. Slated to close at the end of M0.5.

**Sub-milestones.**

- **M0.1 (DONE):** Single-rectangle end-to-end. Crate compiles; SVG figure generated; 14 of 21 named tests passing.
- **M0.2 (DONE):** Corner parcels, sticky back edges (lite), preserve-on-deform, performance instrumentation, `fig_06a/b`. 16 of 21 named tests.
- **M0.3 (DONE):** I3 fix at acute corners, minimum-change deformation, `SplitSegment` preserve, all 21 named tests active.
- **M0.4 (DONE):** Shared-vertex registry. No-drift contract under repeated edits.
- **M0.5 (IN PROGRESS):** Bulletproof overlap (rigorous polygon-polygon test, per-parcel depth caps, polygon-difference cleanup), Voronoi-experiment subdivision for intersections and cul-de-sacs, missing M1 figures, working OBB regularization. Closes M1.0.



**M2 — Interactive test harness**

**Goal.** A clickable UI to place road nodes, drag roads, and see parcels regenerate live, so the API can be stress-tested by a human against arbitrary edits.

**Decision (M2 kick-off).** Implementation = a sibling Rust crate `road_parceling_studio/` using `egui` (immediate-mode UI). Builds for both native (fast iteration) and WASM (browser tab). Reasoning: keeps everything in Rust, no JS-side geometry duplication; `egui` is more productive for tooling than `bevy`.

**Out of scope.** Production game features (zoning, buildings, simulation). The harness is a developer/designer tool.

**M3+ — TBD**

Candidates that come up in conversations:

- Parcel *merge* and *split* operations (agent-driven). Vertex registry already supports the geometry; a parcel-layer DCEL with explicit edge identity is the next data-model upgrade.
- Building system: a real `BuildingFitCheck` implementation, footprint preservation (Q4).
- Game-engine integration (Bevy or similar).
- Multi-threading: `subdivide_all` parallelized per block.

## 15 Open Questions

A running list. Resolved questions migrate to section 16 as Design Decisions and may also leave a note in the journal's session record where they were resolved.

**Q1: Skeleton-based subdivision as a fallback**

Should the straight-skeleton-based subdivision algorithm be implemented as a fallback for blocks where frontage-first produces ugly results? Frontage-first handles 90% of cases cleanly; skeleton handles irregular blocks better but adds significant complexity.

**Status.** Partially resolved (D4 commits to frontage-first as primary). The Voronoi-based experiment in M0.5 may serve the same role as the skeleton fallback; if it does, Q1 closes.

**Q2: Spatial index for affected-parcel lookup**

`apply_road_edit` needs to find all parcels with frontage on a given segment. Linear scan is  $O(n)$  and fine for small cities; an R-tree or grid index becomes necessary at scale. When?

**Tentative.** Ship linear scan in M1 with a documented hot-path comment, swap in `rstar` when the benchmark in section 11 exceeds budget. The shared-vertex registry already uses a spatial hash for vertex lookup, so the scaffolding is partially in place.

**Q3: Block ownership of back edges**

For parcels facing roads on opposite sides of a block, who owns the back edge? Two options: (a) medial line, both deform symmetrically; (b) fixed at creation, one parcel grows while the other shrinks.

**Status.** Resolved (option a) for M0.1; per-edge ray-cast cap delivers symmetric extrusion. Revisited in M0.5 where per-parcel depth caps replace the per-edge variant for tighter no-overlap guarantees.

**Q4: Determinism of regeneration**

When a block is regenerated, the new parcel set differs from the old one. Should the regeneration be biased toward producing parcels that overlap maximally with the old ones, to preserve building footprints opportunistically?

**Tentative.** No for M1; revisit when the building system lands. The shared-vertex registry partially mitigates this — vertices are reused on re-insertion at the same position.

**Q5: Voronoi vs. frontage-first at intersections**

M0.5 introduces an experimental Voronoi-based subdivision for intersection corners and cul-de-sac bulbs. Does it produce visually superior parcels? Should it become the default at high-degree intersections, or stay a fallback?

**Status.** Open; M0.5 implements it as an A/B-able alternative gated behind a `SubdivisionParams::corner_method` flag.

## 16 Design Decisions Index

Canonical record of every locked-in design choice, in order. The journal references these by D-number; this section is the authoritative copy.

**D1, 2026-04-25 – f64 throughout**

Use `glam::DVec2` (f64) crate-wide rather than `Vec2` (f32). Single-precision loses too much accuracy on offset operations at city scales. Cost:  $\sim 2\times$  memory for vertex storage. Worth it.

**D2, 2026-04-25 – DCEL over adjacency list**

Half-edge / DCEL graph representation rather than an adjacency list. Block extraction (face traversal) is the dominant query and is  $O(1)$  per step in DCEL. Cost: more complex insertion / deletion logic.

**D3, 2026-04-25 – Parcels indexed by slotmap key**

`slotmap` for parcel storage rather than `Vec` indexing. Stable IDs are required for I4 (edit persistence): a parcel that survives a road edit must retain its identity for downstream consumers (buildings, agents).

**D4, 2026-04-25 – Frontage-first as primary algorithm**

Implement frontage-first subdivision before any skeleton-based approach. Frontage-first handles the majority of real cases (rectangular blocks, gently curved roads, standard intersections). Skeleton-based subdivision is deferred (Q1).

**D5, 2026-04-25 – BuildingHandle owns its BuildingFitCheck**

`BuildingHandle` wraps a `Box<dyn BuildingFitCheck>` so the deform pipeline can call `fits_in` locally without forcing `apply_road_edit` to take a callback.

**D6, 2026-04-25 – DCEL next/prev rule, explicit form**

`HalfEdge::next` = the *predecessor* (CW neighbor) of `half_edge.twin` in the target vertex's CCW-sorted outgoing list, with wrap. `HalfEdge::prev` = the twin of the *successor* of the half-edge in its origin's list. The standard “CCW after twin” phrasing is ambiguous about rotation direction; this is the unambiguous form. (Derived in section 17.3.3.)

**D7, 2026-04-25 – Polygon::new\_relaxed for block boundaries**

Block polygons may legitimately contain collinear-corner vertices (the `collinear_roads` case). The strict `Polygon::new` rejects collinear triples; `new_relaxed` skips that check. It applies to parcels, not blocks.

**D8, 2026-04-25 – Clippy scope is all, not pedantic**

Crate enables `clippy::all` only. `pedantic` fights numerical-code conventions (single-letter coordinate names, struct-default reassignment) without buying real safety.

**D9, 2026-04-25 – Build-first corner parcels**

At each real corner, the corner parcel is built before the frontage walk; the walk on each adjacent road then starts past the corner's footprint. (Voronoi delete-and-refill alternative considered and rejected: build-first has fewer edge-case surprises.)

**D10, 2026-04-25 – Corner radius is the average frontage width**

Corner parcels extend  $R = \text{params.frontage\_width}$  along the frontage-side adjacent road and `params.depth` along the other. 4-vertex parallelogram for 90° corners; 6-vertex L for  $R \neq \text{depth}$  in some constructions.

**D11, 2026-04-25 – “Real corner” definition**

Corner-parcel routine fires at any block-boundary vertex whose underlying graph node has degree  $\geq 3$ , or degree 2 with a bend angle below 150° (interior  $> 60^\circ$ ). Acute corners are flagged separately and fall back to bisector-clip on adjacent regulars.

**D12, 2026-04-25 – Road width deferred; setback is the placeholder**

Roads stay as zero-width centerlines through M1. The `setback` parameter folds in any visual road-thickness margin downstream consumers want. Revisit when a renderer needs literal road widths.

**D13, 2026-04-25 – Sticky back edges, lite (now superseded by D17)**

Original M0.2 commitment: each parcel stores its full polygon in absolute world coords; deform pipeline only moves frontage vertices. Adjacent parcels' shared back edges happen to coincide. *Superseded by D17*, which makes shared vertices explicit and atomic via the registry.

**D14, 2026-04-25 – Minimum-change deformation (no-op preserve)**

When a road edit doesn't change a road's *line* (only its endpoints shift along it), parcels whose frontage is still inside the new segment are reported as `Untouched` and skip the deformation. Trade-off: strict vertex-by-vertex inverse-restore is no longer guaranteed; centroid-bounded drift is the new contract.

**D15, 2026-04-25 – Bisector-clip at acute corners, not obtuse**

Acute corners (interior  $< 60^\circ$ ) get no corner parcel; instead, regular parcels along the two adjacent edges are bisector-clipped at the corner so their territories stay separated. Obtuse corners ( $\geq 60^\circ$ ) keep their rectangle/parallelogram corner parcel and need no clip.

**D16, 2026-04-25 – SplitSegment preserve on 4-vertex parcels**

On `SplitSegment`, parcels whose frontage is entirely on one side of the split point have their `frontage_road` rebound (no geometric change). Parcels whose frontage spans the split are cut into two parcels along a perpendicular through the split point — only for 4-vertex parcels; higher-vertex parcels fall back to `Condemn`. Buildings stay with the larger half.

**D17, 2026-04-25 – Shared-vertex registry**

`ParcelSet` owns a `SlotMap<VertexId, VertexRecord>` plus a spatial-hash index. On parcel insertion, every polygon vertex is snapped to the registry — existing matches within  $\epsilon_{\text{geom}}$  reuse the same `VertexId`; otherwise a new entry is created. Each `VertexRecord` carries a back-reference list of `(ParcelId, vertex_index)` pairs. This is invariant I8.

**D18, 2026-04-25 – move\_vertex write-through**

`ParcelSet::move_vertex(vid, new_pos)` updates the registry's stored position *and* writes through to every referring parcel's polygon at the recorded index. Adjacent parcels' shared boundaries cannot drift.

### D19, 2026-04-25 – Deform pipeline is propose-then-apply

`deform_parcel_after_road_move` no longer mutates — it returns a list of proposed `(VertexId, new_pos)` moves. The outer loop validates each parcel, collects proposals, then applies them via `move_vertex` after verdicts are in. Conflicting proposals on the same vertex are last-one-wins, but in practice the deform parameterization makes referrers agree by construction.

## 17 System Walkthrough

This section is the “how does this thing actually work” tour. It is written for a graduate-level engineering reader who knows linear algebra and basic algorithmics but is not a computational-geometry specialist. Every formula is derived; every algorithmic step is justified; cross-references point to the file and function in the source where the math is implemented.

### 17.1 Pipeline at 30,000 ft

The crate consumes a planar road graph and produces a set of polygonal parcels with shared boundary vertices. The transformation is staged:

1. **Build the road graph topology.** Roads become a half-edge / DCEL data structure (section 17.3). Faces of the DCEL are blocks.
2. **Extract blocks.** Each bounded face becomes a `Block` polygon with metadata (section 17.4).
3. **Subdivide each block** into parcels via the frontage-first algorithm (section 17.5).
4. **Snap vertices into the shared registry** so adjacent parcels share boundary points (section 17.6).
5. **Edits** (move-node, split-segment, etc.) propose vertex moves; the outer pipeline applies them through the registry, validating each affected parcel (section 17.7).

The two end-to-end traces (sections 17.8 and 17.9) walk a single `subdivide_all` call and a single `apply_road_edit(MoveNode)` call from the public API down to the geometric primitives.

### 17.2 Mathematical primitives

We work in  $\mathbb{R}^2$ . Points are 2-vectors  $\mathbf{p} = (p_x, p_y)$ . The standard vector operations apply: addition, scaling, dot product  $\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y$ , and the 2D “cross product”  $\mathbf{a} \times \mathbf{b} = a_x b_y - a_y b_x$  which is the  $z$ -component of the 3D cross product (a scalar in 2D).

#### 17.2.1 Rotations

The standard 2D rotation matrix is

$$R(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}.$$

At  $\theta = +\pi/2$  (a  $90^\circ$  counter-clockwise rotation) this collapses to

$$R(\pi/2) = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix},$$

so  $R(\pi/2) \cdot (x, y)^T = (-y, x)^T$ . Symbolically: *rotate  $90^\circ$  CCW takes  $(x, y) \mapsto (-y, x)$* . Rotation by  $-\pi/2$  (CW) is  $(x, y) \mapsto (y, -x)$ .

These two rotations are the crate’s workhorses for computing perpendiculars. Given an edge direction  $\mathbf{t} = \mathbf{p}_2 - \mathbf{p}_1$  (normalized), the inward normal of a CCW polygon at this edge is

$$\mathbf{n}_{\text{in}} = R(\pi/2) \mathbf{t} = (-t_y, t_x).$$

This is the direction pointing into the polygon’s interior, because for a CCW polygon the interior lies to the left of each edge as you walk in the edge’s direction, and “left” is exactly  $R(\pi/2)$  applied to the forward direction.

In code (`src/parcel/subdivide.rs`, multiple sites):

```
1 let edge_dir = (q - p) / (q - p).length();
2 let inward = DVec2::new(-edge_dir.y, edge_dir.x);
```

### 17.2.2 Polygon orientation: the shoelace formula

A polygon with vertices  $\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{n-1}$  has *signed area*

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i),$$

where indices are taken modulo  $n$ . This is the *shoelace formula*, named for the cross-multiplication pattern when you write the coordinates in two columns. The sign carries the orientation:  $A > 0$  for CCW vertex order,  $A < 0$  for CW.

**Why this formula works.** Pick any reference point (call it the origin). For each edge  $(\mathbf{p}_i, \mathbf{p}_{i+1})$ , the triangle formed by origin,  $\mathbf{p}_i$ , and  $\mathbf{p}_{i+1}$  has signed area  $\frac{1}{2}(\mathbf{p}_i \times \mathbf{p}_{i+1}) = \frac{1}{2}(x_i y_{i+1} - x_{i+1} y_i)$  (the 2D cross product is twice the signed triangle area). Summing these triangles around a closed polygon, every interior region is covered the same net number of times by triangles of correct sign and zero net times by overlapping pieces of opposite sign — the algebra works out so the sum equals the polygon’s signed area. (This is a consequence of Stokes’ theorem in 2D, but the elementary triangle-sum picture is enough here.)

We use this in `src/geometry/mod.rs`:

```
1 pub fn signed_area(verts: &[DVec2]) -> f64 {
2     let n = verts.len();
3     if n < 3 { return 0.0; }
4     let mut a = 0.0;
5     for i in 0..n {
6         let p = verts[i];
7         let q = verts[(i + 1) % n];
8         a += p.x * q.y - q.x * p.y;
9     }
10    0.5 * a
11 }
```

`Polygon::new` uses the sign to detect CW input and flip it to CCW (so all internal code can assume CCW), and uses the magnitude to detect degenerate (zero-area) polygons.

### 17.2.3 Segment intersection

A segment from  $\mathbf{p}_1$  to  $\mathbf{p}_2$  is the set  $\{\mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1) \mid t \in [0, 1]\}$ . To find where two segments cross we solve the linear system

$$\mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1) = \mathbf{p}_3 + s(\mathbf{p}_4 - \mathbf{p}_3)$$

for  $(t, s) \in [0, 1]^2$ . With  $\mathbf{d}_1 = \mathbf{p}_2 - \mathbf{p}_1$  and  $\mathbf{d}_2 = \mathbf{p}_4 - \mathbf{p}_3$ , applying Cramer's rule gives

$$t = \frac{(\mathbf{p}_3 - \mathbf{p}_1) \times \mathbf{d}_2}{\mathbf{d}_1 \times \mathbf{d}_2}, \quad s = \frac{(\mathbf{p}_3 - \mathbf{p}_1) \times \mathbf{d}_1}{\mathbf{d}_1 \times \mathbf{d}_2}.$$

The denominator  $\mathbf{d}_1 \times \mathbf{d}_2$  is the determinant of the system; it vanishes iff the segments are parallel. The crate treats parallel segments as non-intersecting (they may overlap collinearly, but for our use cases that's not an interesting intersection).

### 17.2.4 Polygon-vs-half-plane clipping (Sutherland–Hodgman)

Given a convex half-plane  $H = \{\mathbf{p} \mid (\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n} \geq 0\}$  (where  $\mathbf{n}$  is the inward normal and  $\mathbf{p}_0$  is any point on the boundary line), and a polygon  $P = (\mathbf{q}_0, \dots, \mathbf{q}_{m-1})$ , the Sutherland–Hodgman algorithm computes  $P \cap H$  in one pass:

1. Initialize an output list, empty.
2. For each edge  $(\mathbf{q}_i, \mathbf{q}_{i+1})$  of  $P$ , classify both endpoints as *inside* ( $d \geq 0$ ) or *outside* ( $d < 0$ ) where  $d = (\mathbf{q} - \mathbf{p}_0) \cdot \mathbf{n}$ :
  - *In–In*: append  $\mathbf{q}_{i+1}$  to output.
  - *In–Out*: compute the intersection of the edge with the half-plane boundary; append it to output. (No endpoint added.)
  - *Out–In*: append the intersection, then  $\mathbf{q}_{i+1}$ .
  - *Out–Out*: append nothing.
3. The output list is the clipped polygon (or empty if everything was outside).

The intersection of edge  $(\mathbf{a}, \mathbf{b})$  with the boundary line is a 1-D linear interpolation. Let  $d_a = (\mathbf{a} - \mathbf{p}_0) \cdot \mathbf{n}$  and  $d_b = (\mathbf{b} - \mathbf{p}_0) \cdot \mathbf{n}$ . Then the intersection lies at

$$t^* = \frac{d_a}{d_a - d_b}$$

along the edge, giving point  $\mathbf{a} + t^*(\mathbf{b} - \mathbf{a})$ . (Derivation: the boundary is  $\{\mathbf{p} \mid (\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n} = 0\}$ . Substituting the parameterized edge and solving for  $t$  yields the formula. The denominator vanishes only when  $d_a = d_b$ , which means the edge is parallel to the boundary — a degenerate case we handle by returning “no intersection”.)

**Why this works for convex clip regions.** A convex region is the intersection of half-planes. If  $H_1, \dots, H_k$  are the inward half-planes for an edge of the convex region, then  $P \cap (H_1 \cap \dots \cap H_k) = (((P \cap H_1) \cap H_2) \cap \dots) \cap H_k$ . Sutherland–Hodgman applies a sequence of half-plane clips, one per edge of the clip region. The crate uses this for two purposes: clipping a regular parcel against a corner bisector, and clipping every parcel against the block boundary's inward half-planes.

In code (`src/geometry/polygon.rs`, `Polygon::clip_half_plane`):

```

1 pub fn clip_half_plane(&self, point: DVec2, inward_normal: DVec2) ->
  Option<Self> {
2     let n = inward_normal.normalize();
3     let inside = |p: DVec2| (p - point).dot(n) >= -EPS_GEOM;
4     let intersect = |a: DVec2, b: DVec2| -> Option<DVec2> {
5         let da = (a - point).dot(n);
6         let db = (b - point).dot(n);
7         let denom = da - db;
8         if denom.abs() < EPS_GEOM { return None; }
9         let t = da / denom;
10        Some(a + (b - a) * t)
11    };
12    // ... iterate edges, emit per the four-case rule above ...
13 }

```

## 17.3 The road graph as a DCEL

### 17.3.1 Why DCEL

A planar graph drawn in the plane partitions the plane into *faces* (the connected components of the complement of the edge set). The road graph’s faces are exactly the city blocks plus one unbounded outer face. We need fast face queries: “what’s the boundary of this block?”, “what’s the block on the other side of this road?”.

A vanilla adjacency-list graph stores edges per vertex, which makes vertex-incidence and edge-existence cheap, but face traversal is  $O(|E|)$  to discover faces from scratch each time. The Doubly-Connected Edge List (DCEL) augments the graph with explicit face structure. Every edge is split into two *half-edges*, one for each direction, and pointer fields on each half-edge support face-walks in  $O(\text{face perimeter})$  regardless of total graph size.

### 17.3.2 Half-edge structure

Each edge is represented as a pair of half-edges. A half-edge  $h$  has:

- **origin**: the vertex it leaves from.
- **twin**: the half-edge for the same physical edge but opposite direction.
- **next**: the next half-edge counterclockwise around the face on  $h$ ’s left.
- **prev**: the inverse of **next**.
- **face**: the face  $h$  bounds (on its left).

The convention we adopt: the face on a half-edge’s *left* (relative to the direction of travel) is the face that half-edge bounds. For a CCW polygon, walking around its boundary with the polygon’s interior on your left, you traverse one half-edge per boundary edge in order.

### 17.3.3 Constructing the DCEL: the next-pointer rule

Suppose we’ve created two half-edges per road, set up their **twin** pointers, and now want to fill in **next** so that following **next** from any half-edge walks the face on its left. The rule is most cleanly derived by asking: “standing at the target vertex of  $h$ , with the face on  $h$ ’s left, what’s the next half-edge that bounds this same face?”



At a vertex  $v$ , the half-edges *leaving*  $v$  have specific compass-headings (angles from  $v$  to their target). Sort them counter-clockwise by angle: in `src/network/graph.rs`, this is `sort_outgoing_by_angle`, which uses `atan2(dir.y, dir.x)`.

Now consider half-edge  $h$ :  $u \rightarrow v$ . Its *twin*  $h^*$  leaves  $v$  heading back toward  $u$ . In the CCW-sorted list of outgoing half-edges at  $v$ ,  $h^*$  sits at some position  $k$ . The face on  $h$ 's left is the face on  $h^*$ 's right, which is the face wedged between  $h^*$  and the half-edge *immediately clockwise* of  $h^*$  in the angular ordering — because the face wraps around  $v$  between two consecutive outgoing half-edges. “Immediately clockwise of  $h^*$  in CCW-sorted order” is the same as “the predecessor of  $h^*$  in the list, with wrap”.

So:

$$\text{next}(h) = \text{predecessor of } \text{twin}(h) \text{ in } v\text{'s CCW-sorted outgoing list, with wrap} \quad (1)$$

where  $v = \text{target of } h$ . The corresponding rule for `prev` is the dual:

$$\text{prev}(h) = \text{twin}(\text{successor of } h \text{ in } u\text{'s CCW-sorted outgoing list, with wrap})$$

where  $u = \text{origin of } h$ .

**Worked example: square.** A unit square with vertices  $A = (0,0)$ ,  $B = (1,0)$ ,  $C = (1,1)$ ,  $D = (0,1)$  has 4 roads  $AB, BC, CD, DA$  and 8 half-edges. At vertex  $A$ , the outgoing half-edges are  $h_{AB}$  (heading  $0^\circ$ ) and  $h_{AD}$  (heading  $90^\circ$ ). CCW-sorted:  $[h_{AB}, h_{AD}]$ .

Take  $h_{DA}$ :  $D \rightarrow A$ . Its twin is  $h_{AD}$  ( $A \rightarrow D$ , heading  $90^\circ$ ). At  $A$ ,  $h_{AD}$  is at position 1; its predecessor with wrap is at position  $0 = h_{AB}$ . So  $\text{next}(h_{DA}) = h_{AB}$ . Continuing:  $\text{next}(h_{AB}) = h_{BC}$ ,  $\text{next}(h_{BC}) = h_{CD}$ ,  $\text{next}(h_{CD}) = h_{DA}$ , closing the cycle around the interior face.

**Why the “CCW after” phrasing was ambiguous.** The literature sometimes states the rule as “ $\text{next}(h)$  is the half-edge immediately CCW after  $\text{twin}(h)$ ”. The trap: “after” depends on which direction around  $v$  you measure. CCW *around the vertex* (= the angle ordering) gives *successor* of twin; CCW *around the face* (= the actual traversal we want) gives *predecessor* of twin. The implementation gets this wrong once and you pay for it: the T-intersection unit test in M0.1 caught this when the T-stem face cycle enclosed the wrong region. (See journal session 1; locked in as D6.)

### 17.3.4 Face extraction

Once `next/prev/twin` are populated, every face is the cycle generated by repeatedly applying `next` from any one of its half-edges. Walk the cycle; the vertex positions trace out the face’s boundary polygon. Use the shoelace formula (section 17.2.2) on the vertex sequence: positive area = bounded face (a block), negative area = unbounded outer face (the exterior).

In `src/network/graph.rs`, `extract_faces`:

```

1 for h_start in edge_keys {
2     if self.half_edges[h_start].face.is_some() { continue; }
3     let mut cycle = Vec::new();
4     let mut h = h_start;
5     loop {
6         cycle.push(h);
7         let next = self.half_edges[h].next;
8         if next == h_start { break; }
9         h = next;

```

```

10     }
11     let pts: Vec<DVec2> = cycle.iter()
12         .map(|&hid| self.nodes[self.half_edges[hid].origin].pos)
13         .collect();
14     let signed = signed_area(&pts);
15     let face_id = self.faces.insert(Face {
16         boundary: h_start,
17         is_exterior: signed < 0.0 || signed.abs() < EPS_GEOM,
18     });
19     for h in cycle { self.half_edges[h].face = Some(face_id); }
20 }

```

## 17.4 Block extraction

A Block is one bounded face of the road DCEL, packaged with metadata for subdivision. `src/network/blocks.rs`:

```

1 pub struct Block {
2     pub(crate) face: FaceId,
3     pub polygon: Polygon, // CCW vertex ring
4     pub(crate) boundary_edges: Vec<HalfEdgeId>, // parallel to polygon
5         vertices
6     pub roads: Vec<RoadId>, // parallel to polygon edges
7 }

```

The polygon is built from the face’s vertex cycle; `Polygon::new_relaxed` is used (not the strict `Polygon::new`) because block boundaries may legitimately contain collinear-corner vertices when adjacent road segments are end-to-end (the `colinear_roads` case — D7).

## 17.5 Frontage-first subdivision

Subdivision runs per block. The high-level flow:

1. Compute interior angles at each block-boundary vertex.
2. Classify each vertex: *real corner* (build a corner parcel) vs. *acute corner* (no corner parcel; bisector-clip adjacent regulars) vs. *smooth continuation* (collinear; no corner treatment).
3. Compute per-edge depth caps (the maximum perpendicular extent of a parcel before it would hit the opposite side of the block).
4. For each real corner, build the corner parcel.
5. For each block boundary edge, walk the segment between corner footprints, place split points, and emit quad parcels.
6. Clip each parcel polygon against the block boundary’s inward half-planes (defense in depth against any geometric oversight).

### 17.5.1 Interior angles and corner classification

At vertex  $v$  with neighbors  $v_{\text{prev}}$  and  $v_{\text{next}}$  in CCW boundary order, define

$$\mathbf{t}_{\text{in}} = \frac{v_{\text{prev}} - v}{\|v_{\text{prev}} - v\|}, \quad \mathbf{t}_{\text{out}} = \frac{v_{\text{next}} - v}{\|v_{\text{next}} - v\|}.$$

$\mathbf{t}_{\text{in}}$  points back along the previous edge;  $\mathbf{t}_{\text{out}}$  points forward along the next edge. The *arriving* direction (which is the previous edge's direction in CCW traversal) is  $\mathbf{t}_{\text{arrive}} = -\mathbf{t}_{\text{in}}$ .

The signed CCW turn from  $\mathbf{t}_{\text{arrive}}$  to  $\mathbf{t}_{\text{out}}$  is

$$\theta_{\text{turn}} = \text{atan2}(\mathbf{t}_{\text{arrive}} \times \mathbf{t}_{\text{out}}, \mathbf{t}_{\text{arrive}} \cdot \mathbf{t}_{\text{out}}).$$

The interior angle at  $v$  for a convex CCW polygon is then  $\theta_{\text{int}} = \pi - \theta_{\text{turn}}$ .

### Classification rules.

- Underlying graph node has degree  $\geq 3$ , OR degree 2 with  $\theta_{\text{int}} < 150^\circ$  ( $\theta_{\text{turn}} > 30^\circ$ ): it's a real corner candidate.
- Of those,  $\theta_{\text{int}} > 60^\circ \rightarrow$  *obtuse real corner*: build a corner parcel.
- $\theta_{\text{int}} \leq 60^\circ \rightarrow$  *acute corner*: no corner parcel, bisector-clip adjacent regulars.
- Everything else (degree 2 with  $\theta_{\text{int}} \geq 150^\circ$ ): smooth continuation, walk through.

(D11; the  $60^\circ$  acute threshold was tuned empirically. Below it, the rectangle corner-parcel construction tries to extrude past the polygon boundary — bisector-clip is the safer fallback.)

#### 17.5.2 Per-edge depth cap (ray-cast)

For each block boundary edge  $e_i$ , cast a ray from the edge midpoint  $\mathbf{m}_i$  in the inward-normal direction  $\mathbf{n}_i$  (section 17.2.1). The first intersection of this ray with another edge  $e_j$  gives a distance  $\rho_{ij}$ . Define the per-edge depth cap as

$$d_{\text{cap}}(e_i) = \frac{1}{2} \min_{j \neq i} \rho_{ij}.$$

The factor of  $\frac{1}{2}$  ensures parcels from  $e_i$  extruding to  $d_{\text{cap}}(e_i)$  and parcels from the closest opposite edge  $e_j$  extruding to  $d_{\text{cap}}(e_j)$  meet (or undershoot) at the midline between them, never overlap. (For convex blocks this is a tight bound under the assumption that the closest opposing edge remains the same across the entire span; for non-convex or strongly-tapered blocks, this is the milestone-0.5 weakness that we patch with per-parcel ray casts.)

Ray-segment intersection: parameterize the ray as  $\mathbf{r}(t) = \mathbf{m}_i + t\mathbf{n}_i$  and the segment  $e_j$  as  $\mathbf{q}(s) = \mathbf{a} + s(\mathbf{b} - \mathbf{a})$  for  $s \in [0, 1]$ . Solving  $\mathbf{r}(t) = \mathbf{q}(s)$  gives a  $2 \times 2$  linear system whose solution (if the determinant is non-singular) yields  $(t, s)$  via Cramer's rule. We accept solutions with  $t > \varepsilon$  and  $s \in [-\varepsilon, 1 + \varepsilon]$ .

Code: `src/parcel/subdivide.rs`, `edge_depth_caps` and `ray_segment_distance`.

#### 17.5.3 Corner parcel construction

At an obtuse real corner  $v$ , the corner parcel is a 4-vertex shape with corners derived as follows. Let  $R = \text{params.frontage\_width}$  (the corner radius, D10) and  $d = \text{params.depth}$ .

We first decide which adjacent road wins frontage — the longer one (with deterministic tie-break on RoadId bits). Two flavors:

- **Frontage on next road.** Vertices:  $v_0 = v$ ,  $v_1 = v + R\mathbf{t}_{\text{out}}$ ,  $v_2 = v_1 + d\mathbf{n}_{\text{out}}$ ,  $v_3 = v + d\mathbf{t}_{\text{in}}$ .
- **Frontage on prev road.** Vertices:  $v_0 = v$ ,  $v_1 = v + d\mathbf{t}_{\text{out}}$ ,  $v_2 = v_1 + R\mathbf{n}_{\text{out}}$ ,  $v_3 = v + R\mathbf{t}_{\text{in}}$ .

where  $\mathbf{n}_{\text{out}} = R(\pi/2)\mathbf{t}_{\text{out}}$  is the inward normal of the next edge (section 17.2.1).

For a 90° corner of a rectangle ( $\mathbf{t}_{\text{in}} \perp \mathbf{t}_{\text{out}}$ , with  $\mathbf{n}_{\text{out}} = \mathbf{t}_{\text{in}}$ ), the corner parcel collapses to an axis-aligned  $R \times d$  rectangle. For non-90° corners (e.g., the 120° corner at the centre of a Y-intersection), it's a parallelogram.

The two flavors differ only in which side carries  $R$  and which carries  $d$ ; the geometry is otherwise symmetric. The frontage edge of the corner parcel is the one adjacent to the chosen winning road; `classify_edges` (`src/parcel/classify.rs`) labels it `Frontage` and the rest `Side/Back`.

#### 17.5.4 The frontage walk

Between corner footprints, regular parcels are emitted by walking the block edge at jittered intervals. For edge  $e_i$  of length  $L$  with start-consume  $c_s$  (the corner footprint at the start vertex) and end-consume  $c_e$  (at the end vertex), let  $t_0 = c_s$  and  $t_{\text{max}} = L - c_e$ . We generate split positions

$$t_k = t_{k-1} + \max(w_{\text{min}}, w_f + \sigma_f \xi_k), \quad \xi_k \sim \text{Uniform}(-1, 1)$$

from a per-road `ChaCha8Rng` until  $t_k$  exceeds  $t_{\text{max}} - w_{\text{min}}/2$ , then append  $t_{\text{max}}$ .

For each consecutive pair  $(t_{k-1}, t_k)$ , define

$$\mathbf{p}_a = \mathbf{p}_{\text{start}} + t_{k-1}\mathbf{t}_{\text{edge}}, \quad \mathbf{p}_b = \mathbf{p}_{\text{start}} + t_k\mathbf{t}_{\text{edge}}.$$

Pick a depth  $d^* = \min(d_p + \sigma_d \eta_k, d_{\text{cap}}(e_i))$  from the same RNG, and form a quad parcel with vertices  $(\mathbf{p}_a, \mathbf{p}_b, \mathbf{p}_b + d^*\mathbf{n}_{\text{in}}, \mathbf{p}_a + d^*\mathbf{n}_{\text{in}})$ .

#### 17.5.5 Block-boundary defense clip

After every parcel polygon is built (corner or regular), it is clipped against the block boundary's inward half-planes via Sutherland–Hodgman (section 17.2.4). For convex blocks this is a no-op for parcels generated correctly; for non-convex blocks (e.g., the Y-intersection sub-blocks, where the block triangle has acute outer corners) it ensures parcels can never extend outside their block face.

In code (`src/parcel/subdivide.rs`, `clip_polygon_to_block`):

```

1 fn clip_polygon_to_block(parcel: &Polygon, block: &Polygon) ->
  Option<Polygon> {
2     let mut current = parcel.clone();
3     let block_verts = block.vertices();
4     for i in 0..block_verts.len() {
5         let a = block_verts[i];
6         let b = block_verts[(i + 1) % block_verts.len()];
7         let edge = b - a;
8         let len = edge.length();
9         if len < EPS_GEOM { continue; }
10        let dir = edge / len;
11        let inward = DVec2::new(-dir.y, dir.x);
12        current = current.clip_half_plane(a, inward)?;
13    }
14    Some(current)
15 }
```

### 17.6 Shared-vertex registry

The registry is the data structure that prevents shared boundaries from drifting across edits. (D17, D18; invariant I8.)

### 17.6.1 Data structure

```

1 pub struct ParcelSet {
2     parcels: SlotMap<ParcelId, Parcel>,
3     vertices: SlotMap<VertexId, VertexRecord>,
4     vertex_grid: HashMap<(i64, i64), Vec<VertexId>>,
5     // ...
6 }
7
8 struct VertexRecord {
9     pos: DVec2,
10    refs: Vec<(ParcelId, usize)>, // each parcel that references this
11                                   vertex,
12                                   // and the index into that parcel's
13                                   polygon ring.
14 }

```

The spatial grid is keyed at  $\varepsilon_{\text{geom}}$  resolution: position  $(x, y)$  maps to cell

$$(\lfloor x/\varepsilon_{\text{geom}} + 0.5 \rfloor, \lfloor y/\varepsilon_{\text{geom}} + 0.5 \rfloor).$$

Two positions within  $\varepsilon_{\text{geom}}$  might fall into the same cell or into adjacent cells; lookup checks the cell and its 8 neighbors.

### 17.6.2 Snap-on-insert

When `ParcelSet::insert(parcel)` is called, every polygon vertex is passed through `find_or_create_vertex(pos)`

```

1 fn find_or_create_vertex(&mut self, pos: DVec2) -> VertexId {
2     let key = vertex_key(pos);
3     for dx in -1..=1 {
4         for dy in -1..=1 {
5             let bucket = (key.0 + dx, key.1 + dy);
6             if let Some(ids) = self.vertex_grid.get(&bucket) {
7                 for &vid in ids {
8                     if let Some(rec) = self.vertices.get(vid) {
9                         if (rec.pos - pos).length_squared() < EPS_GEOM *
10                            EPS_GEOM {
11                             return vid;
12                         }
13                     }
14                 }
15             }
16         }
17     }
18     let id = self.vertices.insert(VertexRecord { pos, refs: Vec::new() });
19     self.vertex_grid.entry(key).or_default().push(id);
20     id
21 }

```

The first hit (if any) within  $\varepsilon_{\text{geom}}$  is reused; otherwise a new `VertexRecord` is created. The parcel's vertex-id list runs parallel to its polygon vertex list.

### 17.6.3 Write-through propagation

`ParcelSet::move_vertex(vid, new_pos)` updates the registry’s stored position and walks every `(parcel_id, vertex_index)` reference to write the new position into the parcel polygon directly:

```

1 pub fn move_vertex(&mut self, vid: VertexId, new_pos: DVec2) {
2     let refs = match self.vertices.get_mut(vid) {
3         Some(r) => { r.pos = new_pos; r.refs.clone() }
4         None => return,
5     };
6     for (pid, idx) in refs {
7         if let Some(p) = self.parcels.get_mut(pid) {
8             p.polygon.set_vertex_unchecked(idx, new_pos);
9         }
10    }
11 }

```

Validation (does the parcel’s polygon stay simple after the move?) is the caller’s responsibility, because validity depends on which combinations of vertices move together. The deform pipeline does this in its propose phase before any move is committed (section 17.7).

## 17.7 Edit pipeline

`apply_road_edit(parcels, graph, edit, params)` is the single public entry for every road-network mutation. The pipeline is propose-then-apply (D19): no parcel is mutated until every affected parcel has been classified and its proposed vertex moves collected.

### 17.7.1 Move-node case

Given `RoadEdit::MoveNode { node, to }`:

1. **Snapshot the graph** (`graph_before = graph.clone()`). We need the old node positions to compute parcel-vertex parameters along the old roads.
2. **Apply the topology mutation** (move the node, re-run `rebuild_topology`). The graph DCEL now reflects the new geometry.
3. **Identify incident roads** (those touching the moved node).
4. **For each parcel on each incident road**, run `deform_parcel_after_road_move(parcel, road, graph_before, graph_after, params)` which is a *pure* function returning a verdict and (if Deformed) a list of proposed `(VertexId, new_pos)` moves.

The verdict logic:

**Untouched check (D14, line-unchanged).** Compute the road’s old direction  $\mathbf{d}_{\text{before}} = (\mathbf{p}_b^{\text{before}} - \mathbf{p}_a^{\text{before}})/L_{\text{before}}$  and new direction  $\mathbf{d}_{\text{after}}$ . The road’s *line* is unchanged iff:

$$|\mathbf{d}_{\text{before}} \times \mathbf{d}_{\text{after}}| < \varepsilon_{\text{angle}}^{\text{small}} \quad \text{and} \quad |(\mathbf{p}_a^{\text{before}} - \mathbf{p}_a^{\text{after}}) \cdot \mathbf{n}_{\text{after}}| < \varepsilon_{\text{geom}}.$$

The first condition says the directions are parallel; the second says the old start point lies on the new line. Combined, these say the line (as an infinite mathematical object) didn’t change — only the segment endpoints shifted along it. If additionally the parcel’s frontage endpoints fall within the new segment’s range  $[0, L_{\text{after}}]$  (computing each as  $\mathbf{p} \cdot \mathbf{d}_{\text{after}} - \mathbf{p}_a^{\text{after}} \cdot \mathbf{d}_{\text{after}}$ ), the parcel’s frontage is still entirely on the new road and the parcel skips the deformation: it is reported as Untouched.

**Frontage projection.** If the line did change, project each of the parcel’s two frontage endpoints. The projection uses the *old* road’s parameter:

$$t_a = \frac{(\mathbf{p}_a^{\text{frontage,old}} - \mathbf{p}_a^{\text{road,before}}) \cdot \mathbf{d}_{\text{before}}}{L_{\text{before}}^2} L_{\text{before}},$$

i.e., the dot product of the parcel-frontage-start-to-road-start vector with the road direction, normalized to the road’s length parameter. (We use  $\mathbf{d}_{\text{before}} L_{\text{before}} = \mathbf{p}_b^{\text{before}} - \mathbf{p}_a^{\text{before}}$  as the road vector and divide by  $L_{\text{before}}^2$  to get the parameter.) Then

$$\mathbf{p}_a^{\text{frontage,new}} = \mathbf{p}_a^{\text{road,after}} + t_a(\mathbf{p}_b^{\text{road,after}} - \mathbf{p}_a^{\text{road,after}}).$$

Same for  $\mathbf{p}_b^{\text{frontage}}$ .

**Validate.** Build a *hypothetical* new polygon with the proposed frontage endpoints (other vertices unchanged). Check:

- The polygon validates as a simple CCW polygon.
- Side-edge rotation: the angle between the old side direction and the new side direction is less than  $\alpha_{\text{max}} = \text{params.max\_side\_rotation}$  (default 30°).
- Area  $\geq A_{\text{min}}$ .
- Frontage length  $\geq w_{\text{min}}$ .

Failures map to **Regenerate** (rotation, simple-polygon failure) or **Condemned** (area, frontage too short). On success, the verdict is **Deformed** and we record the two proposed vertex moves: one for each frontage endpoint, looked up by `parcel.vertex_ids[frontage_index]`.

**Apply.** After all parcels are classified, the outer pipeline does:

1. For each  $(vid, new\_pos) \in \text{proposed moves}$ : `parcels.move_vertex(vid, new_pos)`. This propagates atomically (section 17.6) — a shared frontage-end vertex between two adjacent regulars is updated once, and both polygons see it.
2. For each Deformed parcel, update its `frontage_edge_index` (because `Polygon::new` may have rotated the vertex order during proposal validation if the polygon happened to land in CW orientation).
3. For each Deformed parcel with an attached building, call `building.fits_in(parcel)`; on `false`, evict.
4. Drop Condemned parcels.
5. Regenerate any block where some parcel asked for it.

This propose-then-apply structure (D19) is what guarantees adjacent parcels stay in lockstep: two parcels sharing a vertex both propose the same new position (because the deform parameterization is a function of the vertex’s position alone), so when `move_vertex` fires on the shared `VertexId` the proposals agree.

## 17.8 End-to-end trace: `subdivide_all`

A single call `subdivide_all(&graph, &params)` flows as follows.

1. `params.validate()` (`src/config.rs`) — range-checks every `SubdivisionParams` field, returns `InvalidParams` on failure.

2. If `!graph.topology_valid`, clone the graph and run `rebuild_topology()` on the clone (the public API is `&RoadGraph`, so we can't mutate it). `rebuild_topology()` runs `check_planarity`, `build_half_edges`, `sort_outgoing_by_angle`, `link_next_and_prev`, `extract_faces`.
3. `extract_blocks(graph)` walks the DCEL faces, skipping the exterior, and constructs `Block` structs with their CCW vertex polygons + per-edge `RoadIds`.
4. For each block: `subdivide_block(graph, block, params, block_idx)`.
  - a) Compute `interior_angles` per vertex.
  - b) Classify vertices: `real_corner[i]`, `acute_corner[i]` (section 17.5).
  - c) Compute `depth_caps[i]` for each block edge via ray-cast.
  - d) Decide `frontage_on_next[i]` per real corner (longer adjacent road wins).
  - e) For each real corner, build the corner-parcel polygon; clip against the block boundary; if valid and area  $\geq A_{\min}$ , push to output.
  - f) For each block edge, compute walk bounds (subtracting corner footprints), generate split positions via a deterministic per-road `ChaCha8Rng`, and emit each quad parcel (with bi-sector clip at acute-corner ends, then block clip, then frontage-index recovery).
5. For each generated parcel, `ParcelSet::insert(parcel)` which snaps every polygon vertex to the registry, builds `vertex_ids`, and updates `by_block` and `by_road` indexes.
6. Return `ParcelSet` (or with stats when called via `subdivide_all_with_stats`).

## 17.9 End-to-end trace: `apply_road_edit(MoveNode)`

A single call `apply_road_edit(&mut parcels, &mut graph, RoadEdit::MoveNode {node, to}, &params)` flows as follows.

1. `params.validate()`.
2. `graph_before = graph.clone()`.
3. Apply mutation: `graph.nodes[node].pos = to; graph.topology_valid = false; graph.rebuild_topology()`.
4. `move_node_path(parcels, &graph_before, graph, node, params, &mut outcome)`:
  - a) Collect `incident_roads` (those with `node` as endpoint in the new graph).
  - b) For each road, list parcels on it via `parcels.parcels_on_road(road)`.
  - c) For each parcel, call `deform_parcel_after_road_move` (pure — no mutation):
    - i) Untouched check: line unchanged + frontage in new range  $\rightarrow$  return `Untouched`.
    - ii) Compute proposed new frontage endpoints via parameter projection.
    - iii) Validate hypothetical polygon (rotation, area, frontage length, polygon validity).
    - iv) Return verdict; if `Deformed`, include the two proposed moves and the new frontage edge index.
  - d) Bookkeeping: append parcel-id to one of `outcome.deformed` / `outcome.condemned`; record blocks needing regeneration.
5. Apply phase:



- a) For each  $(vid, new\_pos)$  in proposed moves: `parcels.move_vertex(vid, new_pos)` (writes through the registry).
  - b) Update `frontage_edge_index` on each Deformed parcel.
  - c) Building fit-check loop: evict where `!building.fits_in(parcel)`.
  - d) Drop Condemned parcels.
  - e) For each block in `to_regenerate`: drop its surviving parcels (mark Regenerated), re-run `subdivide_block`, insert each new parcel (mark Created).
6. Return `outcome`.

## A Notation Reference

Symbol	Meaning
$G = (V, E)$	Road graph (planar)
$B$	Block boundary polygon
$B'$	Developable polygon, $B$ offset inward by $d_s$
$d_s$	Road setback distance
$w_f, \sigma_f$	Target frontage width, variance
$d_p, \sigma_d$	Target parcel depth, variance
$\rho$	Regularity slider, $[0, 1]$
$w_{\min}, A_{\min}$	Minimum frontage and area
$\alpha_{\max}$	Maximum side-edge rotation before regeneration
$\varepsilon_{\text{geom}}$	Geometric tolerance, $10^{-6}$ m
$\varepsilon_{\text{area}}$	Area tolerance, $10^{-9}$ m <sup>2</sup>
$\varepsilon_{\text{angle}}$	Angular tolerance, $10^{-4}$ rad
$\mathbf{t}_{\text{in}}, \mathbf{t}_{\text{out}}$	Unit tangents back along prev / forward along next edge at a corner
$\mathbf{n}_{\text{in}}, \mathbf{n}_{\text{out}}$	Inward normals at prev / next edge
$R$	Corner radius (default = $w_f$ )
$d_{\text{cap}}(e_i)$	Per-edge depth cap (section 17.5.2)
$\theta_{\text{int}}$	Interior angle at a polygon vertex
$\theta_{\text{turn}}$	Signed CCW turn at a polygon vertex ( $= \pi - \theta_{\text{int}}$ )