ChatGPT

CODEX 3D Spatial Indexing and Routing System (BCC Lattice) – Functional Specification

Author: Michael A. McLarney (with ChatGPT-5)
Maintainer: CODEX Engineering Team (GitHub Repository)

Overview and Goals

The CODEX 3D Spatial Indexing and Routing System is an open-source library and toolset for representing 3D space as a multiresolution grid based on a **body-centered cubic (BCC) lattice** with truncated octahedral cells. The goal is to enable efficient spatial analysis, indexing, and pathfinding in three dimensions at multiple scales. Key objectives include:

- **Uniform 3D Tiling:** Use truncated octahedron cells that tile 3D space without gaps [1], providing nearly isotropic adjacency (14 neighbors per cell) for more natural path planning than a standard cubic grid. Each truncated octahedron has 14 faces (8 hexagons, 6 squares) [1], meaning each cell touches 14 others, allowing movement in a rich set of directions.

- **Multiscale Hierarchy:** Define a hierarchical spatial index (similar in spirit to quadtrees/octrees but on a BCC lattice) that supports **8:1 refinement** at each level. This enables seamless zooming in/out, aggregation of data across scales, and coarse-to-fine analysis or routing.

- **Robust Cell Identification:** Provide a 128-bit (default) and 256-bit (extended) **cell ID** format encoding the cell's position, scale (resolution), and other metadata. IDs are designed for uniqueness, lexicographic ordering (for spatial sorting), and compactness. Human-friendly encoding (Bech32m) with checksums will be used to avoid errors in text representation [2] [3].

- **First-Class Routing & Aggregation:** Make pathfinding and data aggregation core features. The system will support A* and other graph search algorithms on the cell network, including dynamic replanning for changing environments. It will also support efficient aggregation queries (e.g. summing or averaging values in a region or rolling up fine cells to coarse cells).

- **Scalable Rust Implementation:** Emphasize clean, safe, and scalable implementation in Rust. This includes leveraging concurrency (Rayon), SIMD where beneficial, and designing the code for maintainability and clarity. Comprehensive tests (property-based and unit tests) and benchmarks will ensure correctness and performance from the start.

Use Cases: This system is applicable to 3D mapping (e.g. occupancy grids for robotics or drones), geospatial volumetric analysis (e.g. environmental data cubes), path planning in 3D (autonomous UAV routing, gaming, or indoor navigation), and any scenario requiring a multiscale partitioning of 3D space with efficient neighbor relationships and indexing.

The following specification details the mathematical foundation, ID scheme, core operations, data handling, routing algorithms, I/O, and implementation considerations to guide the CODEX engineering team in building version 1.0 of the system.

Mathematical & Structural Foundation

BCC Lattice and Truncated Octahedral Tiling

At the heart of the system is the **body-centered cubic (BCC) lattice** in $\mathbb{R}^3$. A BCC lattice can be visualized as a standard cubic lattice with an extra point at the center of each cube. Formally, one way to describe the lattice is: start with all integer grid points ($\mathbb{Z}^3$) and **retain only those points whose coordinates have identical parity** (either all even or all odd) [4]. This "checkerboard" selection yields two interpenetrating cubic sub-lattices offset by half a cube. Equivalently, one set of points can be at coordinates (even, even, even), and the other at (odd, odd, odd) in integer terms, representing the two offset grids. These points collectively form the BCC lattice.

Each BCC lattice point is surrounded by a characteristic polyhedral region of space closer to that point than any other lattice point – i.e. the **Voronoi cell** of the lattice. For a BCC lattice, the Voronoi cell is a *truncated octahedron* [5]. The truncated octahedron is a 14-faced Archimedean solid (8 regular hexagon faces and 6 square faces) [1]. Crucially, **truncated octahedra tile 3D space perfectly** (they are *parallelohedra*, like cubes, that can fill space by translation) [1]. This means we can partition all of 3D space into these cells with no overlaps or gaps – an ideal property for a spatial index.

Cell Adjacency (Neighbors): Because the truncated octahedron cell has 14 faces, each cell will have up to 14 adjacent cells (sharing a face). In the BCC lattice, each lattice point has 14 equally nearest neighboring points [6], reflecting the same count. Of these 14 neighbors, 8 correspond to the hexagonal faces (which are oriented diagonally relative to the cubic axes) and 6 correspond to the square faces (aligned with axis directions). All 14 are considered **first-ring neighbors** (adjacent cells sharing a face). The geometry ensures these neighbors are uniformly distributed around the cell, giving the grid more isotropic connectivity than a simple cubic grid (which has only 6 face-neighbors). This isotropy is advantageous for routing, since movement can proceed roughly equally in many directions.

Developer Note: The BCC lattice is known to be an optimal sampling lattice for isotropic 3D functions, requiring ~29% fewer samples than a cubic lattice for equal fidelity [7] [8]. While our focus is indexing and routing (not signal processing), this optimality means the truncated octahedral grid is a very efficient representation of space.

Parity and Coordinate Systems

We define an integer coordinate system (x, y, z) for lattice points such that one parity of coordinates (e.g. all even) corresponds to one sub-lattice and the opposite parity (all odd) corresponds to the other. This parity condition (identical parity of x,y,z) succinctly captures membership in the BCC lattice [4]. For example:

- (x,y,z) = (2,4,0) – all even, so this is one lattice point.
- (x,y,z) = (5,7,3) – all odd, so this is another lattice point (the "body-centered" point relative to the even lattice).
- (1,2,3) – mixed parity (one even, two odd) would **not** be a lattice point at the chosen resolution.

We can choose the origin and scaling such that the *even-parity* points correspond to the “primary” sub-lattice and the *odd-parity* points to the interstitial sub-lattice (or vice versa). In one convenient physical interpretation, the even-parity points could lie at integer coordinates in some unit, and the odd-parity points lie at offsets of (½,½,½) in those units. In this setup, the lattice spacing (distance between neighboring lattice points of the same parity) is 1 in those units, and the nearest neighbors are the opposite parity points at a distance of $\sqrt{}3/2$.

Parity Bit: Internally, each cell can be considered as having a parity flag (0 or 1) indicating which sub-lattice it belongs to. However, we may not need to store this explicitly if using the coordinate parity rule. The sum `x + y + z` being even or odd can be used to derive parity on the fly (for a valid lattice coordinate, `x+y+z` will always be even for one sub-lattice and odd for the other [9]). Parity is important for certain operations (like neighbor calculations) as described later.

Logical vs Physical Cell Containment

The hierarchical grid system defines a parent-child relationship between cells at different resolutions. It’s important to distinguish **logical containment** from **physical (geometric) containment**:

- **Logical containment:** In the indexing hierarchy, each cell at a given resolution *divides* into a fixed number of smaller cells at the next higher resolution. By design, in this system **each cell logically has 8 children** at the next finer resolution (hence “8:1 refinement”). This is analogous to each coarse cell being subdivided into 8 smaller subcells (like an octree subdivision). We assign child cells based on subdividing the lattice coordinates (and thus the space) in half along each dimension.

- **Physical containment:** Unlike a cube, a truncated octahedron is not self-similar under simple halving of edge length and axis-aligned subdivision. Eight smaller truncated octahedra will have the same total volume as one larger truncated octahedron, but they cannot be arranged to exactly fill the parent cell’s shape without some offset. In other words, the children do **not lie neatly inside the exact boundaries of the parent cell** – the parent’s volume is a different shape than the union of its children’s volumes. However, the children are defined such that their *union covers the same region of space* as the parent cell did (plus possibly portions that were formerly attributed to adjacent parent cells due to how the lattice shifts).

In practice, we treat the parent-child relationship as an **indexing construct**: a child’s index is formed by refining the parent’s index (splitting coordinate intervals in half). This will ensure that *every point in space maps to a unique cell at each resolution*. Some child cells will extend slightly beyond the geometric boundary of the parent’s cell and into what was neighboring space at coarser level – but those areas were then part of neighbor parent cells which themselves split. The net effect is that at finer resolution the space is re-tessellated entirely by smaller truncated octahedra. Every coarse cell can be thought of as *approximately* containing its 8 children, but exact geometry boundaries differ. We will document this clearly to avoid confusion in usage.

Developer Guidance: Emphasize the logical hierarchy in code. For example, obtaining the parent of a cell can be done by simple bit-shifts or division of coordinates by 2 (dropping the least significant bit of each coordinate). This is straightforward even if geometrically the parent’s polyhedron doesn’t strictly envelop the child’s. The logical hierarchy is still extremely useful for indexing and aggregation, as it partitions space consistently across scales.

Scale, Resolution, and Sizing

We use *resolution* (an integer level) to denote the scale of cells: - **Resolution 0** could be defined as the coarsest level in use (for instance, cells on the order of the largest dimension of interest). We might define resolution 0 such that one cell covers a very large region (if modeling a planet, maybe one cell covers a huge volume). In practice, resolution 0 might also be used to denote the *base lattice spacing*. For flexibility, we will treat resolution 0 as the base BCC lattice with some unit edge length (configurable). - Increasing the resolution by 1 (to resolution 1, 2, …) **halves the linear dimensions** of each cell, resulting in cells that are half the edge length and 1/8 the volume of cells one level coarser. Each increment adds one more level of detail, increasing coordinate precision by a factor of 2.

Because of the halved edge length, **each cell at resolution *n*** corresponds to **8 cells at resolution *n+1*** (logical children). We maintain this as an invariant of the hierarchy.

We will sometimes refer to *scale* in a physical sense – for example, if resolution 0 cells have an approximate circumradius or “radius” R, then resolution 1 cells have radius R/2, etc. But in most of the spec, *resolution number* is the primary way to denote scale.

Coordinate Scaling by Resolution: We embed the resolution into the coordinate system by an implicit scale factor. We can imagine that at resolution *n*, all BCC lattice coordinates are scaled by `2^n` relative to some base length. Equivalently, one can define *integer coordinates at resolution n* that relate to *integer coordinates at resolution 0* by a factor of 2^n. For example, if a point has coordinates (x0, y0, z0) at resolution 0 (in base units), then at resolution *n* the “same location” might be represented as (x_n, y_n, z_n) = (x0 * 2^n, y0 * 2^n, z0 * 2^n). In practice, we don’t explicitly scale physical positions; instead, we note that as resolution increases, the lattice gains new intermediate points.

Defining Base Length: We choose a convenient physical length for the edge of the truncated octahedron at resolution 0. This could be 1.0 in some coordinate unit, or tied to a real-world measure (for example, 1km). It will often be simplest to take the base lattice distance (distance between two even-parity lattice points along an axis) as 2 units, so that an adjacent body-centered point is at (1,1,1) offset (which in physical distance is $\sqrt{3}$ since each offset is 1 unit in x,y,z – but by choosing 2 as base step, the center-to-corner distance is normalized nicely). The exact choice can be hidden behind the API, with conversions if needed.

Precision Consideration: With each resolution step adding a factor of 2 to coordinate values, integer coordinates can grow large at high resolutions. The 128-bit ID design will accommodate a generous range of resolution and coordinate magnitude (detailed below). For extreme cases (very fine resolution or very large coordinates), the exponent field in the ID provides additional scaling headroom beyond fixed bits.

Hierarchical Cell ID System

Each cell in the lattice at any resolution is assigned a **globally unique identifier (cell ID)**. This ID encodes the key properties of the cell: what “frame” of reference it’s in, which resolution (scale), and the cell’s lattice coordinates at that resolution. It also includes some extra bits for flags or error-checking. The ID is the fundamental handle for cells, used as keys in databases, file storage, or passing through APIs.

ID Bit Layout and Sizes

We define two primary binary formats for cell IDs: - **128-bit ID (u128):** The default format for most uses. It packs all necessary information for typical applications. - **256-bit ID (u256):** An extended format for cases needing either extremely high resolution or additional custom data/flags. The extended format might allocate extra bits to coordinate precision or additional metadata. - *(Optional)* **512-bit path ID (u512):** A very large ID form intended to explicitly encode the entire *subdivision path* from a root cell down to a very fine cell. This could be used for extremely deep hierarchies or for denoting precise positions with a path string. (More on path encoding below.)

All IDs are unsigned integers under the hood, allowing lexicographic ordering to correspond to numeric ordering (useful for range queries).

Proposed 128-bit Layout: We allocate the 128 bits into fields as follows (subject to adjustment during implementation):

- **Frame (8 bits):** Identifier of the coordinate frame or reference system. This allows multiple disjoint lattices or different geodetic datums. For example, frame `0` could mean a default Euclidean XYZ grid in meters, frame `1` might indicate a WGS84 Earth-centric coordinate mapping (if we adapt this system to Earth), etc. In many cases we'll use frame 0 (default), but this field ensures extensibility for future multi-frame support (e.g., planetary data, map projections, etc.). 8 bits gives up to 256 frames, which is plenty.

- **Resolution (8 bits):** The resolution level of the cell (0 to 255). This field indicates how fine the cell is. 8 bits covers up to 255 levels of subdivision, which is far beyond practical needs (2^255 refinement!). In typical use, we expect resolutions maybe up to 20–30 for very fine grids (and likely much less). We reserve a full byte for clarity and possible repurposing of bits (e.g., using some high bits as special flags, see exponent below).

- **Exponent / Scale Factor (4 bits):** *Arbitrary scaling exponent.* This is an extra field to support extremely large scales or very fine detail beyond the normal resolution range. It effectively acts as a multiplier or offset on the resolution. For example, we might interpret the actual resolution as `resolution + (exponent * N)` or use the exponent to indicate that coordinate values should be interpreted with an extra 2^exponent factor. The exact usage can be defined such that the 8-bit resolution handles the typical range and the exponent extends it. Another interpretation is using exponent as a *shift* for coordinates (like scientific notation: coordinates * 2^exponent). In any case, these 4 bits can allow representing cells at scales even more extreme without exhausting the resolution field or coordinate bits. If not used, this field can be zero. (In 128-bit IDs, not all 4 bits may be needed; some could be reserved for future flags too.)

- **Coordinates (3 × 24 bits = 72 bits):** The (x, y, z) coordinates of the cell's lattice point at the given resolution, offset from some origin. 24 bits per axis gives coordinate values in [0, 16,777,215] for each axis (if unsigned interpretation) or we may prefer signed coordinates centered at 0 (e.g., using 24-bit two's complement for –8,388,608 to +8,388,607 range). Whether we use signed or unsigned depends on how we choose the origin of our coordinate system. Likely we will use a **signed coordinate system** centered at the origin of the frame (0,0,0) = some reference point (perhaps an Earth-centered origin or just an arbitrary 0). Using signed coordinates is convenient for areas around an origin, while unsigned might be used if we only operate in one quadrant. Given 72 bits total, we can cover extremely large spaces (over 8 million units in each direction at base resolution, and even more at lower resolutions when interpreting coordinates coarsely). In extended 256-bit IDs, these coordinate fields can be extended (e.g., 48 bits each axis for even more range and precision).

- **Flags (4 bits):** A small field for boolean flags or category markers at the cell level. For example, we could use a couple of bits here to mark the cell as having certain properties (like reserved bits to indicate if the cell is on a boundary of a dataset, or if this ID encodes something special like a "hexagon" or alternate shape in some extension mode). In the initial version, we anticipate using some of these bits internally (like to mark an "invalid" placeholder ID, or to indicate if an ID has a path extension). We will define any flag usage clearly, and leave unused bits zeroed.

- **Checksum (8 bits):** A small checksum or Cyclic Redundancy Check (CRC) over the ID bits (could be 8-bit CRC). This is mainly for internal validation if needed. However, since our primary *external* representation uses Bech32m (which has strong error detection built-in [2]), this internal checksum is optional. We might use it in binary contexts to quickly detect misassigned IDs or mixing IDs from different frames, etc. Another use might be to guard against simple bitflip errors in storage. We can decide to implement a CRC-8 or similar scheme (or repurpose these bits in future if redundant).

Byte Alignment: The above fields sum to 8+8+4+72+4+8 = 104 bits, leaving 24 bits unused in the 128-bit layout. Those unused bits can be left reserved (for future expansion of fields such as more coordinates or more flags). Alternatively, we might expand coordinate bits to use more of the space (e.g., 25 bits each for x,y,z = 75 bits, and add the leftover to flags or exponent). The exact bit allocation will be finalized during implementation, balancing the need for range vs. keeping some growth room.

256-bit ID Variation: The 256-bit ID will follow the same conceptual layout but with larger allotments where needed: - Frame and basic resolution fields can remain the same size (as 128 bits is already generous for them). - Coordinates can be expanded significantly (e.g., 48 bits per axis or more) allowing extremely large domains or extremely fine indexing at deep resolutions. - Exponent field could be kept or slightly extended if needed (or use some extra bits to effectively increase resolution range). - Flags could be expanded to include more layer identifiers or feature bits (for example, a flag for "high precision mode" or indicating if this ID is part of a particular dataset). - The checksum can be extended to 16 bits for stronger error checking, or we might include a full 32-bit CRC if 256-bit IDs are used primarily in binary (noting that Bech32m can also encode 256-bit strings with its own robust checksum if we use that externally).

The 256-bit format essentially future-proofs the system. We expect most applications won't require 256-bit IDs unless dealing with **extreme scales** (e.g., mapping the entire solar system at cm resolution) or adding rich metadata directly into the ID (which usually is not necessary when a database can store metadata separately).

Encoding and Decoding (Bech32m and Binary)

For external use (APIs, humans reading/writing IDs, config files, etc.), cell IDs will be represented as **Bech32m strings**. Bech32m is chosen because it produces compact alphanumeric strings with excellent error-detection properties [2] [3]. Specifically, Bech32/Bech32m uses a restricted character set (32 characters, all lowercase by design), and includes a strong checksum that can catch almost any single or multiple-character error (and even pinpoint errors) [2]. This makes our cell IDs human-friendly: if someone types or copies an ID, the chance of an undetected mistake is extremely low. We will follow BIP-350 (Bech32m) standard for checksum calculation, which improves error detection resilience [3].

Bech32m Format: A Bech32m string consists of a **human-readable prefix** (HRP), a separator `1`, followed by the data encoded in base32, and a 6-character checksum at the end. For CODEX cell IDs, we will define a unique prefix such as `"cx3d"` or `"cdx"` (to be decided). For example, an encoded ID might look like:

```
cx3d1x5yfk...a7mz (etc)
```

The prefix (`cx3d`) can indicate "CODEX 3D" and possibly the frame if needed (though frame is also in data bits). We will ensure the prefix ends with `1` when forming the full string per Bech32m rules.

Internally, encoding an ID means taking the 128-bit binary, splitting into 5-bit groups, and mapping to the Bech32 alphabet, then appending the checksum. We'll use a Rust crate (e.g., `rust-bech32` which supports Bech32m [10]) to handle this, to avoid writing our own encoding unless needed for customization.

Decoding will perform the reverse: verify the checksum and prefix, extract the data bits, and reconstruct the 128-bit or 256-bit value. We'll validate that the HRP (prefix) matches expected (to avoid confusion with other Bech32 uses) and possibly check that the frame from the bits aligns with the prefix if we encode frame in HRP.

Binary Representation: In binary form (e.g., in memory or writing to a binary file or database), the ID is just a 16-byte or 32-byte sequence. We need to decide endianness for serialization (likely big-endian/ network order so that lexicographic byte sorting equals numeric sorting – this is typical for such IDs). If little-endian is more convenient for bitfield implementation, we can convert to big-endian when serializing externally. The system will provide utilities to convert IDs to/from bytes or hex for debugging.

ID Parsing and Validation: Functions will be provided to: - Create an ID from frame, resolution, and coordinates (with optional flags) – handling range checks (e.g., ensure coordinates fit in allotted bits). - Extract fields from an ID (get frame, resolution, etc.) easily. - Parse a Bech32m ID string into a binary ID and vice versa. - Validate an ID (check checksum, verify that parity of coordinates is consistent if needed, etc.). The checksum bits (if used) can be verified, and for additional assurance, we might recompute the parity and ensure it matches expectation (though parity might not need encoding because coordinates inherently carry it).

Developer Note: Use property-based tests (via proptest) to ensure round-trip integrity of encoding/decoding – generate random field values, encode to string, decode back, and check equality. Also test common failure modes: e.g., strings with 1 character changed should fail checksum, etc.

Hierarchical Path Encoding (u512)

While the primary ID encodes the absolute position of a cell, we also consider an optional scheme to encode the *ancestry path* of a cell from a top-level root down to itself. This can be seen as similar to a Morton code or quadkey in 3D. Each cell has 8 children; thus, each refinement step could be encoded by a 3-

bit child index (since 8 = 2^3, we need 3 bits to uniquely label each child among siblings). A sequence of such 3-bit indices forms a path from a coarse cell to a fine cell.

In a 512-bit value, we could pack up to ~170 refinement steps (since 170 * 3 ≈ 510 bits) which is overkill for any realistic depth (170 levels of resolution). More practically, we might allocate some bits for a starting "base cell" index and then subsequent groups of 3 bits for child choices. This approach is reminiscent of how the H3 library encodes a hexagon's ancestry or how geohash strings work, but tailored to our 3D grid.

Use of Path IDs: We do not anticipate needing path-IDs in the initial version, but the design leaves room for this concept: - They can be useful for *spatially sorting cells* (path IDs impose a Z-order or similar space-filling curve order). - They could allow variable resolution specification in one ID (just by the number of bits used). - If we want to address sub-cell precision (point locations within a cell), a path code could pinpoint an arbitrary sub-division (like infinite zoom) as far as needed by extending bits.

If implemented, path encoding would be an alternate mode (likely indicated by a flag in the ID). For now, we primarily document this as a future extension. The first version will focus on fixed-size IDs as above.

Developer Guidance: When implementing the ID bit layout, keep the code modular – e.g., a struct with bitfields or methods to get/set parts. This will help if we adjust bit allocations. Also, consider adding debug formatting for IDs that prints them in a human-readable form (maybe decode to string or at least show fields) to ease debugging.

Core Lattice Operations

With the lattice defined and cells indexed, the system will provide a suite of core operations on the lattice. These are fundamental geometric or topological queries that algorithms (and users) will rely on. All operations should handle arbitrary resolution (taking the resolution into account as needed) and be efficient, as they may be called millions of times in large analyses or pathfinding.

Neighbor Lookup (14 Neighbors)

Each cell has up to 14 neighbors (adjacent cells sharing a face). We need to be able to find all neighbors of a given cell ID quickly. This can be derived from the cell's lattice coordinates.

Neighbor Coordinate Offsets: Based on the BCC lattice structure, the neighbor offsets can be enumerated in coordinate space. Let's denote a cell at coordinates (i, j, k) (in the integer coordinate system of its resolution). Neighbors can be categorized by whether they lie on the same parity sub-lattice or the opposite:

- **Opposite Parity Neighbors (8 neighbors):** These are the neighbors that lie on the other sub-lattice (if our cell is even-parity, these neighbors are odd-parity, and vice versa). In coordinates, an opposite parity neighbor can be reached by offsetting each coordinate by ±1. In fact, the 8 combinations of adding or subtracting 1 to each of (i, j, k) yield the 8 neighboring lattice points on the other parity. For example, from (i, j, k) -> (i+1, j+1, k+1) is one, (i-1, j-1, k-1) is another, and (i+1, j+1, k-1), etc. Any combination of ±1 on each axis (no zeros) flips the parity (since you add an odd number to each coordinate) and results in a neighbor [6]. These correspond to the truncated octahedron's hexagon-face neighbors, which are the closer ones.

- **Same Parity Neighbors (6 neighbors):** These neighbors lie on the same parity sub-lattice. To remain on the same parity, you must change coordinates by an even amount overall. The neighbors in this category can be reached by offsetting **two axes by ±1 and the third by ∓1**, which effectively is the above case (that actually flips parity, so disregard that), or by offsetting one axis by ±2 and keeping the others the same. In simpler terms: the 6 neighbors on the same parity can be given by **(±2, 0, 0)**, **(0, ±2, 0)**, **(0, 0, ±2)** from (i, j, k). For example, (i+2, j, k) yields a neighbor, as does (i-2, j, k), (i, j+2, k), etc. These moves keep the parity identical (since you're adding an even number to one coordinate and 0 to others, the parity condition `x+y+z mod 2` remains unchanged) [9]. These correspond to the square-face neighbors of the truncated octahedron – slightly farther in distance than the diagonal ones, but still directly adjacent.

Thus, we have 14 offset vectors to obtain neighbors. At boundaries (if we have any finite bounds), some neighbors might fall outside the domain – the code should handle that (e.g., if we limit coordinates or if using an Earth wrapping model, though currently we consider Euclidean space without wraparound).

Computing Neighbors: To get neighbor IDs, one can: 1. Decode the target cell's (x,y,z) and resolution from the ID. 2. For each of the 14 offset vectors (pre-defined constant list), add the offset to (x,y,z). 3. If the offset changes resolution (it shouldn't; these offsets are within the same resolution grid), keep resolution same. 4. If the neighbor coordinate goes out of allowed range (if we have some bounded grid), skip or handle appropriately. In an unbounded lattice, all 14 exist. 5. Construct the neighbor cell IDs from the new coordinate and same resolution (and same frame).

We will provide this as a function `neighbors(cell_id) -> Vec<cell_id>`. The neighbor ordering can be fixed (perhaps list opposite-parity ones first, then same-parity, or some canonical ordering). For many algorithms (like A*), the order doesn't matter for correctness but could for performance (we may choose to order by, say, ascending ID or a pattern).

Optimization: We can optimize neighbor computation by noticing that adding ±1 or ±2 to coordinates can be done on the 3 integers with some bit twiddling. But the straightforward approach is fine. A potential micro-optimization is to precompute the neighbor offsets in terms of bit patterns if interleaving or using a single 64-bit to store coordinates; this is not too important initially. Another is to use branchless techniques to compute parity and apply appropriate offsets.

Parity Checks: Interestingly, if we take an (x,y,z) and apply an opposite parity offset (±1 each), the result's parity should indeed be flipped compared to original. We might assert that as a sanity check in dev tests. The correctness of neighbors essentially comes from the BCC definition: two lattice points are neighbors (share a face) if and only if their difference is of form (±1, ±1, ±1) or (±2,0,0) etc. We can double-check using known geometry if needed (it matches known results where 8 nearest and 6 next-nearest fill the faces).

Memory of Neighbors: For pathfinding, we will call neighbor lookup extremely often. We should ensure it is efficient and possibly *allocates no heap memory*. Instead of returning a vector, we might return an array of 14 IDs or use a small array on stack. We could also provide an iterator. In Rust, returning a fixed-size array of length 14 is possible (maybe as `[ID;14]`), or we return an array of Option IDs if some neighbors missing (but we typically have all 14 in infinite grid). A small `Vec` might be fine given 14 elements, but we can avoid allocation by passing in a callback or using an output slice. This is a low-level performance detail we can refine after initial implementation.

k-Rings and Shells

Building on neighbor queries, a **k-ring** of a cell is the set of all cells within *k* steps (graph distance ≤ k) from the origin cell. Similarly, a **k-shell** (or "ring" in some contexts) is the set of cells at *exactly* distance k. These are useful for queries like “find all cells within radius k” or for spreading activation in the grid.

We will implement functions to collect k-rings and k-shells: - `k_ring(cell_id, k) -> Vec<cell_id>`: all cells with graph distance ≤ k (including the cell itself, presumably, for k=0). - `k_shell(cell_id, k) -> Vec<cell_id>`: all cells with graph distance = k (the “edge” of the ring).

Approach: We can perform a breadth-first search (BFS) from the origin cell outwards k steps. Since k is likely to be relatively small in usage (a few steps), and branching factor is 14, this is manageable. We will use a queue (or iterative layered expansion). To avoid duplicates, we need to mark visited cells – using perhaps a hash set. However, for k-rings it might be feasible to derive a pattern mathematically (for small k, one could derive coordinate bounds), but BFS is general and clear.

We should be mindful that the number of cells grows roughly like (branching factor)^k for large k (14^k), but for moderate k it's fine. (We don’t expect users to ask for extremely large k-rings without other constraints.)

We will ensure the functions avoid revisiting cells. For k-shell specifically, one can run BFS to exactly k layers and return the last layer’s results.

Performance: If k is small, BFS overhead is trivial. If k is large (say k=100), the number of cells in the ring is enormous (roughly a sphere of radius 100 in this lattice ~ on order of (distance)^3 ~ million cells). That could be heavy. For such cases, if needed, we could provide a more direct computation or a bounded BFS. But likely k will be small in typical usage (like neighborhood of 2 or 3 for smoothing filters, etc.). We’ll document that large k could be expensive.

Parent/Child Mapping (Resolution Refinement 8:1)

As part of the hierarchical index, we need to efficiently compute the parent of a given cell (at resolution n to resolution n-1), and the children of a cell (at resolution n to n+1). This is analogous to moving up or down an octree.

Parent ID: For a cell at resolution `r > 0`, the parent at resolution `r-1` can be found by essentially reversing the coordinate doubling. If the child’s coordinates are `(x_r, y_r, z_r)` at resolution r (these would be integers where parity rule applies at that level), the parent’s coordinates `(x_{r-1}, y_{r-1}, z_{r-1})` can be obtained by **integer division by 2** (or right shifting by 1 bit if considering binary). In other words:

```
x_{parent} = floor(x_r / 2)
y_{parent} = floor(y_r / 2)
z_{parent} = floor(z_r / 2)
```

This effectively drops the least significant bit of each coordinate, which were specifying the child's position within the parent. We must also determine the parent's parity – it should naturally come out correct from the division since if child was on a certain sub-lattice, the parent will be on one of the sub-lattices at the coarser level (the parity flips or not depending on the path parity count, but it doesn't matter explicitly; the parent coordinate itself will satisfy the parity rule at the coarser level because of how the BCC lattice nests: the BCC property is self-similar under doubling—odd coordinates become even or vice versa accordingly).

We will implement `parent(cell_id)` that returns the parent cell's ID (or None if the cell is at resolution 0 and thus has no parent). This is simply manipulating the coordinate bits and decrementing the resolution field.

Children IDs: Given a cell at resolution `r` (with coordinates `(x, y, z)` at that level), we define its 8 children at resolution `r+1` as those cells whose coordinates `(x_c, y_c, z_c)` at level r+1 satisfy:

```
x_c = 2*x + δx
y_c = 2*y + δy
z_c = 2*z + δz
```

where δx, δy, δz ∈ {0, 1}. There are 2^3 = 8 combinations of (δx, δy, δz). Each combination yields one child coordinate. In effect, we are taking the parent's coordinate and appending one additional binary digit to each axis coordinate (the δ bits).

For example, if parent coordinate is (10, 4, 7) at res r, its children coordinates at r+1 will be: - (20+0, 8+0, 14+0) = (20, 8, 14) - (20+0, 8+0, 14+1) = (20, 8, 15) - (20+0, 8+1, 14+0) = (20, 9, 14) - (20+0, 8+1, 14+1) = (20, 9, 15) - (20+1, 8+0, 14+0) = (21, 8, 14) - (20+1, 8+0, 14+1) = (21, 8, 15) - (20+1, 8+1, 14+0) = (21, 9, 14) - (20+1, 8+1, 14+1) = (21, 9, 15)

These 8 are the logical children IDs. Note that half of these will have even parity sums and half odd (the parent was either even or odd parity at res r; at res r+1 both parity types appear among the children). But that's expected since the finer lattice has double the points.

We will implement `children(cell_id)` to return the 8 child IDs. This involves pulling the parent's coord and resolution, then computing the 8 combos as above, and packing into IDs with resolution+1.

Edge cases: If we are at max resolution (e.g., 255 if using 8-bit field), then asking for children might not make sense or be representable in the same ID format. We could return an error or empty list in that case. However, given such a deep level is unlikely, we can simply handle it gracefully (e.g., return empty to indicate no finer representation available in current encoding).

Logical Volume Partition: As noted before, these children cover space that corresponds to the parent's space *plus* possibly some adjacency adjustments. However, by generating them via coordinate doubling, we ensure a **consistent grid**. We should confirm that every child of all parents tiles the entire space at resolution r+1 exactly once (which it should by the nature of refining the lattice).

Cell Traversal (3D Bresenham or Ray Marching)

A common operation is to traverse through a sequence of cells along a straight line between two points (or between two cell centers). For example, given a start and end point in space, we might want to list all cells that a straight line between them intersects. This is useful for line-of-sight, collision checking, or simply drawing a line on the grid.

We will implement a 3D line traversal algorithm akin to a 3D Bresenham's algorithm (also known as a 3D Digital Differential Analyzer, 3D-DDA [11]). The algorithm incrementally steps through the grid one cell at a time, always choosing the next cell that the line enters.

Approach Outline: - Input: two points in space (could be given as cell IDs or as continuous coordinates). If given as cell IDs, we'll use their center coordinates as start/end. If given as arbitrary coordinates, we first determine the starting cell (e.g., using a point-in-cell query). - Determine the parametric equation of the line: **direction vector** (dx, dy, dz) and current position. - Use a DDA algorithm: compute the parametric distances to the next cell boundary in each of x, y, z directions. Essentially, this requires computing how far one must travel along the line to cross a vertical plane, a horizontal plane, etc., from the current position. Amanatides & Woo's algorithm (commonly used in ray casting) precomputes `tMax` for each axis (the distance along the ray to the first grid boundary in that axis) and `tDelta` (distance along ray to cross one cell in that axis) [12] . Then, at each step, move in the direction of the smallest tMax, and increment that tMax by tDelta for that axis. - We need to adapt the algorithm for a BCC grid. The standard DDA algorithm assumes an *axis-aligned grid of cubes*. Our grid is not axis-aligned in the same way; however, we can transform the problem. One strategy: - Map the BCC lattice to a skewed coordinate system: We know a BCC lattice can be represented by a basis of vectors (for instance, one basis for BCC is (1,1,0), (1,0,1), (0,1,1) in some normalized units). We might convert the line into that basis where the grid becomes axis-aligned in the transformed space (this is a bit complex). - Simpler: perform the traversal in the underlying cubic lattice of double resolution. Because a BCC lattice can be seen as every other point in a cubic lattice, we might treat the problem in a refined cubic grid where each BCC cell corresponds to two or so steps. This is potentially easier: figure out which truncated octahedron the line is in at each step by stepping through an equivalent process in the dual grid (the Delaunay triangulation or similar).

However, to keep scope manageable, we can initially implement a **voxel traversal on a fine cubic grid** that approximates the cells, or just directly sample along the line at small intervals to pick cells (less efficient but straightforward). A proper BCC-specific line traversal might be an advanced feature.

For version 1, a **simpler approach**: - Compute the sequence of lattice points the line goes nearest. We can discretize the line by some resolution and then map points to cell IDs. For example, sample the line at fine increments (like step through param t from 0 to 1 in small increments, find the cell for each sample, and add unique cells). This is brute force and could oversample but guarantees we find all intersections. If the line is not too long or resolution not too high, this might be acceptable. - Alternatively, do a BFS flood-fill from start towards end but that is complicated to ensure minimal path.

Given time, we aim to implement a true grid traversal algorithm: 1. Determine the starting cell and ending cell. 2. If start == end, trivial. 3. Otherwise, determine stepping direction for each coordinate (sign of dx, dy, dz). 4. Compute initial `tMaxX, tMaxY, tMaxZ` = distance to first boundary on each axis. Compute `tDeltaX, tDeltaY, tDeltaZ` = distance between crossings (which is basically |step_length/Δcoordinate|). 5. Iterate: at each step, whichever of tMaxX, tMaxY, tMaxZ is smallest indicates crossing that axis boundary first; step to the adjacent cell in that axis direction, and add the corresponding tDelta to that axis's tMax. Record the new cell. 6. Continue until the target cell is reached (or criteria if moving beyond).

Adapting to truncated octahedra is tricky because the boundaries between cells are not aligned to global axes, but the above algorithm might still work if we treat the lattice's coordinate system (with half-steps) carefully. Each truncated octahedron face lies either on a plane like x+y = constant (for hex faces) or on a plane x = constant (for square faces). We might incorporate both types of boundary in the check. This is complex to derive generally, so the initial implementation might choose to approximate or iterate in small steps.

Outcome: We will provide a function `trace_line(start_point, end_point, resolution) -> Vec<cell_id>` which returns the list of cell IDs intersected. It will be documented that this function currently may approximate and that a more exact method is planned. We will ensure the list has no duplicates and is in traversal order.

Developer Note: This is a candidate for future optimization. The current focus is to have a working implementation (even if not the most optimal) that can be tested and improved. The line traversal should be verified by simple scenarios (like purely horizontal, vertical, or 45° lines) to ensure the cells visited make sense.

3D Region Filling (Polyfill Operations)

Spatial analysis often involves selecting all cells that lie within a certain geometric region (volume). We will support polyfill operations for basic shapes: - Axis-aligned boxes (rectangular prisms) - Cylinders (likely aligned with one axis, e.g., vertical cylinders) - Extruded polygons (a 2D polygon extended between two z-values, essentially a prism with polygonal base) - Possibly spheres or ellipsoids (not mentioned explicitly, but could be future).

The operation will output the set of cell IDs whose cell volumes intersect the given shape. This is analogous to the "polyfill" function in H3 (which fills a polygon with hex cells).

Approach for Boxes: If the box is axis-aligned in the coordinate system of the frame, this is simplest. We can determine the range of coordinates that overlap the box and iterate: - Determine the min and max coordinates (x_min…x_max, etc.) in the lattice that intersect the box. Because cells are convex and space-filling, one way is to find the min cell that contains the box's min corner and max cell that contains the box's max corner in each dimension. - Then loop through that range and select cells whose center lies inside the box. Or more rigorously, check if the cell polyhedron intersects the box (but that's heavy; using center point is a heuristic that might miss edge cases where a cell is only barely in the box). - Given truncated octahedra have a diameter (farthest distance across) of certain length, using center inclusion might be fine if box is large relative to cell. If the box is small or oddly positioned, better to do an exact check. But exact polyhedron intersection is complex. - We might approximate by including any cell whose center is within the shape or whose any corner of bounding box is within the shape.

General Polyfill via Sampling: A general reliable method is: - For a given shape, find its bounding box. - Within that bounding box, sample a fine grid of points (e.g. at higher resolution than target) and mark which cell those points fall into (using point->cell lookup). This will over-sample and ensure all touched cells are caught. However, this could be slow for big volumes.

Alternatively, use geometric tests: - For each candidate cell, test if the shape intersects the cell polyhedron. Intersection of convex polyhedra (like truncated octa and a box or cylinder) can be done via separating axis theorem. But that's probably too heavy for now.

We can optimize for specific shapes: - For an axis-aligned rectangular prism: The condition for a cell to intersect it might be broken into coordinate constraints. Because truncated octa is somewhat spherical, likely if the center of cell is within half a cell diameter of the box, it's intersecting.

- For a vertical cylinder (say aligned with z-axis): we can polyfill by iterating cells layer by layer in z. For each z-slice (maybe at certain height ranges), determine the circle of radius R intersection and fill roughly a hex pattern.

- Extruded polygon (common use-case: e.g. a building footprint with height): we can first take the polygon in horizontal plane, use a 2D polygon fill approach to get cells in each layer (like treat it like multiple horizontal H3 polyfill but in our grid for each relevant z). That means:

- Project all cell centers in the z-range onto the horizontal plane.
- Check which are inside the polygon (point-in-poly test).
- If inside and within height bounds, include those cells.

This can be done per relevant z coordinate layer.

For initial version, we might implement polyfill of a *convex* region in a simpler way: - If the region is convex (like box, cylinder) and reasonably aligned, we can optimize by scanning coordinate ranges. - If concave or arbitrary polygons, we handle as collection of convex or via point test loops.

We will likely focus on **GeoJSON polygon support** meaning an arbitrary lat/lon polygon extruded between altitudes. That requires converting lat/lon to our XYZ frame (likely Earth-centered ECEF coordinates for 3D). That gets complex, but if frame indicates something like ECEF, we can take polygon, assign altitude range, then fill.

Functionality: We will have `polyfill(region, resolution) -> Vec<cell_id>`, where `region` can be specified as: - A bounding box (min/max X,Y,Z or center+sizes). - A cylinder (center line and radius). - A polygon with min/max Z. - Possibly point+radius (sphere). We'll likely implement multiple functions or one that branches based on `region` type.

Output considerations: The output could be a very large list (if region is big at fine resolution). We should be careful with memory. Perhaps we stream results or allow filtering. But for now, returning a Vec is fine.

Parallelization: Polyfilling a large region is embarrassingly parallel (we could partition the space and compute in parallel). We will definitely leverage Rayon for potentially large fills. For example, if filling a polygon across many lat/lon points, we can split by row or by vertical slice.

We will incorporate parallel processing in the implementation for heavy cases, while ensuring determinism (the output list ordering might not be guaranteed unless we sort at end). Probably sorting cell IDs at the end (which could be expensive if millions of cells) – maybe not by default. The user may not need them sorted, or if they do, they can sort. We should clarify this.

Developer Note: Use existing computational geometry where possible (maybe use `geo` crate or similar for point-in-polygon tests if dealing with lat/lon or planar polygons). Also consider memory: if a region covers millions of cells, returning a huge vector may be heavy. We might provide an iterator interface or write directly to a file as an output option in CLI.

Data Layers and Cell Attributes

Beyond spatial indexing, the system will support associating **application data** with each cell (or with many cells). This could include things like occupancy (free/obstacle), terrain elevation, temperature, categories, etc. We treat these as **data layers** attached to cells.

Attribute Storage Model

Each cell can have a set of attributes, each of a specified type: - Numeric types: integer (various sizes), floating-point (f32, f64), boolean (as 0/1 or bit flags), categorical (which can be represented as small enums or integers). - Potentially more complex types like strings or blobs could be allowed, but for spatial analysis, we emphasize numeric/categorical data.

We will likely store attributes in a **columnar fashion** for efficiency: - For each attribute field, maintain an array or map keyed by cell ID to the attribute value. - Alternatively, for sparse data, use a hash map from cell ID to a small struct of all attributes for that cell. - Since many 3D grids are sparse (e.g., only store obstacles, or only store occupied cells in a largely empty space), a sparse structure might be prudent. However, we can encapsulate this decision behind an API.

For version 1, we might implement a simple **in-memory database**: - A global map (like `HashMap<CellID, CellDataStruct>`) where `CellDataStruct` can hold various typed fields (perhaps using an enum for each field or just multiple parallel maps, one per field). - Provide APIs to set/get attribute values for a cell, create new layers, etc.

Because the specification explicitly lists operations like sum, mean, etc., we consider that we will often need to iterate through many cells' values, so storing values contiguously (arrays) can help with performance (cache locality, SIMD). However, building a dense array of size equal to all possible cell IDs is infeasible (the space is huge/infinite). We can restrict the domain (e.g., by region of interest) or use sparse arrays keyed by index offsets.

Conclusion: We likely implement a **Layer** abstraction: - A `Layer` has a name, a data type, and stores values for some set of cells. - For dense-ish usage (like a region of interest fully covered by cells), we might use a `Vec` where index corresponds to some linearization of cell coordinates (complicated for 3D). - For general usage, we use a `HashMap` or `BTreeMap` from cell ID to value. - There can be multiple layers (like one for "elevation", one for "temperature", etc.) in memory.

We should also consider memory usage and concurrency: writing to layers from multiple threads could require locks. If doing heavy writes (like ingesting a big dataset), we can partition data or use concurrent map structures (or fill in single thread and then use read-only for multithread ops).

Flag/Mask Bits in Cells

We expect certain boolean attributes to be common – e.g., whether a cell is *blocked*, *no-fly zone*, *water cell*, *boundary*, etc. To optimize, we can allocate a bitmask field in each cell data (or even inside the ID, but we decided ID's flag bits likely not for this). More straightforward is to have a special **flags layer** where each cell's value is an integer bitmask of flags: - For instance, define bits: bit0 = blocked (obstacle), bit1 = no-fly, bit2 = water, bit3 = boundary, etc. (We can define up to, say, 16 or 32 such flags.) - If a flag is not relevant, it remains 0 in those bits.

This approach compresses several booleans into one field and could speed up certain operations (like quickly checking if a cell is traversable by doing `if flags & BLOCKED_BIT != 0`).

We will define a default interpretation for a few common flag bits as above. The user can choose to use them or not. Additional flags could be user-defined in higher bits of that field if needed.

Aggregation Operations

Aggregation is a first-class capability: being able to combine values from multiple cells (either at the same resolution or across different resolutions). The system will support a variety of aggregation functions: - **Count:** number of cells or points in a set. - **Sum:** sum of values (e.g., sum of population across cells). - **Mean:** average of values. - **Median:** 50th percentile of values. - **Quantiles:** general percentiles or specific (e.g., 25%, 75% for IQR). - **Mode:** most frequent categorical value. - **Min/Max:** minimum or maximum. - **Standard Deviation/Variance:** for distribution spread.

These can be applied in a few contexts: 1. **Aggregation of child cells to parent:** e.g., compute the sum or average of an attribute from all 8 children of a coarse cell. This can be used to *roll up* data to a coarser grid (for instance, computing coarser occupancy or average elevation). Some aggregates like count or sum aggregate naturally; others like mean or median we define suitably (mean of parent = average of children, median of parent = median of children values, etc.). 2. **Aggregation of arbitrary set of cells:** e.g., summing all cells within a region query, or average value along a path, etc. 3. **Point-in-cell aggregation:** If we have raw point data (e.g., sensor readings or point clouds), we may want to aggregate those into cell values. For instance, count how many points fall into each cell (point density), or sum of point weights, etc. This is essentially binning data into cells – which is a spatial aggregation from points to cells.

We will design an API for aggregation. Options: - Provide a high-level `aggregate(layer, cells_set, agg_type)` function that given a list of cell IDs and an operation returns the aggregated result. - Provide methods on layers like `layer.aggregate(cells, Agg::Mean)` etc. - For child->parent specifically, we might have a function `rollup(parent_cell, layer, agg_type)` that fetches its children's values and aggregates them.

Because this could be done for many parents, we will likely implement a batch roll-up: e.g., `rollup_all(layer, source_resolution, target_resolution, agg)` that goes through all cells at source_resolution (or all children of every parent) and computes parent layer values in one go. This can be parallelized.

Example: To create a coarse layer from a fine layer, one might call `downsample(layer_fine, layer_coarse, agg=mean)` which takes each parent cell and assigns the mean of its 8 children's `layer_fine` values to the corresponding entry in `layer_coarse`.

We also consider weighted aggregations (maybe not needed in v1, but quantiles require sorting or selection algorithms).

For quantiles and median, if we have a large number of values, we won't want to sort each time from scratch. We might use streaming algorithms or order-statistic trees. But given moderate child count (only 8 children in hierarchy case), median is simple (just sort 8 values). For arbitrary region of many cells, a median or quantile might be more challenging; but it can be done by collecting all values and selecting.

Mode is for categorical data (we might just use a frequency count from a hash map of categories among the cells).

The system will implement these in a straightforward way first (collect values in a Vec and then compute, which is fine for hundreds of values; for millions, need better, but likely region queries will use simpler metrics like sum or mean for performance).

We will also provide incremental update support in algorithms like pathfinding (for cost) or interactive filtering – but that's more advanced. For now, static aggregation queries are fine.

Children Roll-up and Parent Downsampling

We touched on roll-up above. We emphasize that because each parent has exactly 8 children (except at boundary if domain is clipped), we can efficiently aggregate up the pyramid. We might build pyramids of data (like mipmaps) for faster queries at coarse resolution.

Design: We can include a utility to automatically generate all coarser resolutions for a given fine layer. For example, if we have a layer at resolution 10 fully populated, we can compute layers at resolution 9,8,...,0 by successive roll-ups (with chosen aggregate function for numeric layers). This could be done recursively or iteratively. We might default to sum or average depending on context.

This would be useful for multi-resolution pathfinding (coarse cost maps) and visualization at different zooms.

Point-In-Cell Aggregation

To integrate external data (like a list of points with values), we provide functions to bin them into cell values: - Given a list of points (each with, say, a numeric measurement), and a resolution, we determine the cell for each point and accumulate (e.g., count points per cell, or sum values per cell). This is effectively a spatial join operation. - We will implement a function `rasterize_points(points, resolution, agg)` that returns a layer (or a map of cell->value). If `agg` is count, it counts points per cell; if sum, sums their property; if mean, we track sum and count then divide; if other, more logic.

This operation should be efficient – possibly sorting points by cell or using a hash map insertion. We will use the neighbor-finding logic (or a direct coordinate transform) to get cell ID from point position. For each point: - Compute its cell ID by determining which BCC lattice point it's closest to (for that resolution's lattice). There is a way to compute that by rounding coordinates in the appropriate basis. We might do a small search around floor(x)… because BCC not aligned, but likely using parity rule and rounding half offsets. - Increment the accumulator for that cell in a hash map or array.

We can accelerate by parallelizing if points list is huge, using Rayon: split points list, each thread accumulates to local map, then merge maps. Merging might be heavy but there are techniques (maybe use DashMap crate for concurrent map insertion).

Routing and Pathfinding

One of the core features is efficient routing through the cell graph. Each cell is a node, neighbors define edges. We will implement pathfinding algorithms with pluggable cost functions.

A* Search (and Variants)

We will implement the A *algorithm for shortest path on the grid graph. Key elements: - Node = Cell ID. State includes cell and maybe accumulated cost. - Neighbors = 14 neighbor cells (as determined by neighbor lookup). - Cost: Each neighbor transition has a cost, computed by a user-defined cost function (default could be the Euclidean distance between cell centers or simply 1 for uniform cost). - Heuristic**: We will use the straight-line distance (Euclidean) between the current cell center and the goal cell center as the default heuristic. This is admissible (does not overestimate actual path cost) because the shortest path in the grid cannot be shorter than the direct straight line distance (especially since our grid allows fairly free movement directions). If we incorporate more complex costs (like different weights), the heuristic must be adapted accordingly (e.g., multiply distance by a minimum cost-per-distance factor).

A* Implementation: We'll use a typical priority queue (min-heap) to pick the next cell with lowest f = g + h. We maintain a map of visited or best-known cost for each cell to avoid reprocessing with higher cost. Since the grid can be very large or infinite, we rely on heuristics to focus on the route corridor; A* effectively explores a ball around the start towards goal.

We should be careful with memory – e.g., if searching globally without obstacles, it could expand a lot. But in practice, with obstacles or on a finite map, it's manageable. Also, we can provide an option to limit search radius or cost.

Bidirectional A*: For long distances, we will implement* bidirectional A***, which runs two simultaneous searches: one forward from the start, one backward from the goal, and stops when they meet. This roughly halves the search depth and can drastically reduce explored nodes. We need to ensure consistent heuristics (usually the same heuristic works both ways) and that the meeting criterion is correct. A typical approach is to run until the sum of g_start(current) + g_goal(current) exceeds the best found path or the frontiers meet.

We will provide this as an option or a separate function `astar_bidirectional(start, goal, cost_fn)`.

Memory and performance: We'll use Rust's `BinaryHeap` for open set. Perhaps switching to `priority_queue` crate or others if needed. We will tune by benchmarking pathfinding on large grids.

Multi-Resolution Path Planning (Coarse-to-Fine)

To handle very large domains or to speed up pathfinding when high resolution detail exists, we support a hierarchical approach: 1. **Coarse Planning:** Plan a path on a lower-resolution grid first. E.g., if we have resolution 10 detail but the area is huge, we can temporarily use resolution 5 (much fewer cells) to get an approximate route. 2. **Refinement:** Take the coarse path and refine it by examining the corresponding finer cells. For example, for each pair of adjacent coarse cells on the path, we can treat those as a "corridor" and run a finer A *constrained to that corridor. Alternatively, we can project the coarse path down (each coarse cell corresponds to some set of fine cells – maybe choose the central line through them or the sequence of child cells connecting). 3. Possibly iterate: do hierarchical pathfinding where we plan at multiple levels (like HPA* or similar algorithms do).

A simpler implementation: - Determine an appropriate coarse resolution (maybe based on distance: e.g., if start and goal are far, start coarse). - Compute path at that coarse level (coarse cells likely represent a large area, ignoring small obstacles if any – though we could treat a coarse cell blocked if any of its finer cells are blocked heavily, or use a cost like "number of blocked subcells"). - Once we have a coarse route (a sequence of coarse cell IDs), we "walk" this route with a finer search: essentially, go from start to goal, but only allow expanding into children of those coarse route cells (or near them) as intermediate waypoints.

Alternatively, treat the coarse path as waypoints: for each consecutive pair on the coarse path, pick boundary points and plan at fine resolution just between those boundaries (with the rest of grid open). The union of those fine segments forms the full path.

We will design the API such that the user can request a multi-res plan. Possibly: `route_multires(start, goal, base_res, coarse_res, cost_fn)` where base_res is fine detail and coarse_res < base_res is used for initial planning. The function would: - Aggregate or approximate costs at coarse res (e.g., mark coarse cell as traversable if any fine cell path through it, or assign it a cost equal to average or min cost of fine cells). - Run A *at coarse res. - Take that sequence, then project it down to base res by refining each step (maybe subdividing linearly or doing mini A* from one coarse cell's center to the next coarse cell's center at fine res, ensuring to stay roughly within those two coarse cells footprint). - Return the fine path.

This is complex to implement generally. For v1, we might implement a simpler version: If the grid is very uniform and obstacles not considered at coarse level, coarse path might go through "blocked" coarse cells since coarse cell may contain obstacles. One might have to inflate costs or treat coarse cell blocked if any subcell blocked (which might overly constrain). A better approach: let coarse cost represent the *best* possible fine path through that coarse cell – but finding that is as hard as the original problem albeit smaller.

Time is limited, so at minimum, we will stub this functionality with caution: The spec calls for multi-resolution coarse-to-fine planning as a feature, so we'll include a method, but note its limitations and perhaps recommend certain uses (like static environments where coarse just gives rough direction and fine does the heavy lifting near obstacles).

We can mention research like hierarchical pathfinding (HPA*) where they cluster cells into “entrances” and refine, but implementing that fully may be beyond initial version.

We will likely implement a basic version: plan coarse ignoring obstacles or using partial info, then refine.

Developer Guidance: Multi-res is tricky; ensure that the fine path still is valid (if coarse path went through an area that’s actually blocked in fine, the refinement needs to detour – possibly deviating from coarse route). This can be solved by planning fine in a local window around the coarse path, or by adjusting coarse path with finer checks. D* Lite might help here too (like plan coarse, then as you “execute” it in fine, adjust when hitting issues).

We should mark this feature experimental in v1.

Cost Plugin System

Different applications need different cost metrics for pathfinding: - Pure distance (shortest path geometrically). - Travel time (which might weight vertical movement differently, or penalize certain cells). - Energy or risk cost (maybe entering certain areas has a high cost). - Turn angle cost: e.g., if we want smoother paths, we penalize sharp turns (so the pathfinding will prefer gentler curves even if slightly longer).

To enable this, we design a **cost plugin interface**. In Rust, this could be a trait like:

```
trait CostFn {
    fn cost(&self, current: CellID, neighbor: CellID, layer_data: &Layers) ->
f32;
    fn heuristic(&self, current: CellID, goal: CellID) -> f32;
}
```

Where an implementation can define the cost of stepping into a neighbor (possibly using cell attributes or flags) and provide a heuristic estimate to goal.

For example: - A simple cost function implementation might just return a constant (1) for any move, and heuristic as Euclidean distance. - A distance cost might compute the actual 3D distance between cell centers (accounting that some neighbor moves are longer than others – recall in BCC, the diagonal neighbors are at distance ~0.866 * d and axis neighbors at distance 1 * d if we normalized base distances; we can compute precisely and use that as base cost). - A “no-fly” zone cost function could check a layer flag: if the neighbor cell has `NO_FLY` flag, you could either: - make cost infinite (so A *will avoid it entirely unless no alternative), - or add a very large penalty (so it will only use it if absolutely necessary). - A “terrain” cost function might use a layer of terrain difficulty and add cost proportional to that cell’s difficulty value. - To incorporate turn angles: The cost function can be stateful if we include previous move direction. But A* as typically formulated doesn’t explicitly carry direction. We can fudge it by including direction as part of the state (increasing state space by factor of neighbors). This is more advanced. Alternatively, we can post-process the path to smooth it. For now, we might not fully implement turn angle penalization unless easy (one hack: when computing cost to a neighbor, if we know the parent cell in the A *tree and that parent's parent, you can approximate turn and add cost – but A* then isn't purely Markovian state). Likely skip explicit turn cost in v1, or implement as simple "prefer straight" smoothing after.

Our implementation will focus on allowing the cost function to at least depend on the neighbor distance and maybe neighbor cell attributes.

We'll supply a default cost function (distance + high penalty for blocked cells, etc.).

Dynamic Replanning (D *Lite / LPA*)

In many scenarios, the environment can change (e.g., new obstacles appear or move). Recomputing A *from scratch each time can be expensive if the changes are minor. D* Lite **(Dynamic A*) and **LPA*** (Lifelong Planning A*) are algorithms that reuse previous search results to update the path efficiently* [13]*. They essentially run a modified version of A* that can update distances when edge costs change, without restarting completely [14] [13].

We plan to include support for replanning by integrating either D *Lite or LPA*: - We maintain the search tree (or graph of costs) after the initial pathfind. - When a change occurs (e.g., a cell's cost goes up because it became blocked, or a new neighbor opened), we update the costs and propagate changes through the search frontier. - D* Lite in particular is well-suited for robot moving through unknown space: it plans from goal to start and as the robot moves and discovers obstacles, it reuses the old plan, adjusting the necessary part local to the changes [14] [13].

Implementation details: - Likely maintain a priority queue of nodes to update (the "open list" persists). - Each cell has g and rhs values (LPA* concept of one-step lookahead cost). - When cost changes, update rhs and push affected nodes.

We might not fully implement D* Lite in v1 due to complexity, but we will design the system so that the pathfinding module can accept incremental updates: For example:

```
let planner = PathPlanner::new(start, goal, cost_fn);
let initial_path = planner.search();
...
// later, if some cell costs updated:
planner.update_edge(cell, neighbor, new_cost);
let new_path = planner.replan();
```

This would be the conceptual API.

If time permits, we implement a simplified LPA *(which D* Lite is based on). If not, we document that this is planned and ensure the architecture can accommodate it (like storing search trees).

We definitely will mention using D* Lite for dynamic environments [15], as it's proven in robotics (Mars rovers etc. used it [15]).

Edge Pruning and Cost Modifiers

Not all neighbor moves may be allowed or desirable. We can incorporate rules to **prune edges** from the neighbor graph: - For example, perhaps vertical up moves are not allowed for some agents, or beyond a slope threshold, etc. Or if a cell is completely blocked (like an obstacle), we don't include moves into it at all. - Another example: If the agent has a size, maybe we disallow moves that squeeze through single-cell gaps, etc., which would be higher-level logic.

The system can provide hooks or parameters for pruning: - A global flag or function that given current cell and neighbor cell can decide if that edge is traversable or should be skipped. - We might integrate this with the cost function (cost = ∞ means effectively pruned).

It's simplest to handle by cost function returning infinity (or a sentinel) for forbidden transitions and A* logic skipping those.

Cost modifiers are adjustments to the base cost of moving to a neighbor: - e.g., a slight cost to encourage straight lines can be applied based on direction change (we discussed). - Perhaps a cost multiplier for going from ground to air (if mixing modes). - Or a different cost for entering a certain region (like we can pre-mark some cells with extra cost, e.g., water has high cost for ground vehicle, etc.).

We ensure the cost function and data layers suffice to express these. Possibly we'll create some *preset cost functions* such as: - `EuclideanCost` (cost = distance) - `ManhattanCost` (for debugging or alternate metric) - `TerrainCost(layer_name)` (cost = distance * terrain_factor from a layer) - `AvoidAreaCost(layer_flag, penalty)` (cost = distance + penalty if cell has a certain flag) - etc.

These can be combined or the user can implement the trait for complete control.

Developer Note: We will carefully test the cost handling to ensure heuristics remain admissible. If user supplies a weird cost (like turn angle penalty which is not purely distance-based), the straight-line distance might under- or over-estimate. We may instruct the user that if they use non-distance costs, the default heuristic might not be valid (or they should supply a matching heuristic in the trait). In worst case, we fall back to Dijkstra (A* with h=0) to guarantee optimality at cost of performance.

I/O and CLI Tools

The system will provide command-line interface (CLI) utilities for common operations, as well as support various file formats for input/output to integrate with other tools.

CLI Tools

We envision a `codex3d` (name tentative) CLI with subcommands for various functionalities:

- **Cell ID Utilities:**

- `codex3d id-from-coord <x> <y> <z> -r <res>`: Convert a raw coordinate to the cell ID at given resolution. This uses the point-in-cell mapping (finding nearest lattice point).
- `codex3d id-to-coord <cell_id>`: Decode an ID to get frame, resolution, and lattice coordinates or possibly the center coordinate in physical units.
- `codex3d neighbors <cell_id>`: Print the 14 neighbor IDs of the given cell.
- `codex3d children <cell_id>` / `parent <cell_id>`: For debugging hierarchy.

- **Neighborhood and Region Queries:**

- `codex3d k-ring <cell_id> -k <K>`: Output the IDs of all cells within K steps (graph distance) of the given cell (this could be a lot, maybe support limiting or summary).
- `codex3d polyfill <geojson_file> -r <res>`: Read a region (polygon or region) from a GeoJSON input and output the list of cell IDs that cover it. Possibly options to output to a file, or statistics.
- `codex3d query <region spec> -agg <agg_type> -layer <layer_name> -res <res>`: e.g., aggregate a layer over a region on the fly.

- **Pathfinding:**

- `codex3d route <sx,sy,sz> <gx,gy,gz> -r <res> [--cost <cost_params>]`: Compute a path from start coord to goal coord at given resolution (or allow start/goal given as cell IDs or coordinates). Options to specify which cost model to use (like `--cost distance` or `--cost time layer=wind` etc.), and maybe output the path as a sequence of cell IDs, or as GeoJSON LineString, etc.
- Possibly `codex3d route-multires <start> <goal> ...` to trigger coarse-to-fine planning.

- **Aggregation and Data:**

- `codex3d load-points <file> -layer <name> -r <res> -agg <count|sum|...>`: Import a list of points (from CSV or GeoJSON with points) and aggregate into a grid layer. For example, load all earthquake locations and count per cell.
- `codex3d aggregate -layer <name> -region <spec> -agg <func>`: Compute an aggregate on a specified sub-region.
- `codex3d rollup -layer <finelayer> -out <coarserlayer> -agg <func>`: Perform hierarchical roll-up (could be multiple levels until top or one level).

The CLI will help test functionality and demonstrate usage. Each command will utilize the library's API underneath.

We will design CLI output to be readable or machine-parseable: - Possibly output JSON or simple text lines of IDs. For polygon fill, output could be a GeoJSON FeatureCollection of cell center points or cell boundary outlines (though outlines of truncated octahedra is complicated to output; might skip that). - For route, output could be coordinates of path or cell IDs.

We should allow writing output to files (with format choice maybe CSV, JSON).

File Formats

We plan to support a few file formats for saving and loading data:

- **CBOR (Concise Binary Object Representation):** A compact binary JSON-like format. We can use CBOR to serialize things like a set of cell IDs or a layer data. For example, a polyfill result or an aggregated data cube could be stored as CBOR for efficient reload. We might design a CBOR schema e.g.:

  ```
  {
    "frame": 0,
    "resolution": 12,
    "cells": [ [x1,y1,z1, value1], [x2,y2,z2, value2], ... ]
  }
  ```

  But likely we'll have a custom serialization via Rust's Serde such that writing a layer or list of IDs to CBOR is straightforward.

CBOR is handy because it's simple and has libraries. It's not columnar but for moderately sized outputs it's fine.

- **Parquet/Arrow:** For large-scale or analytical use, Parquet (columnar on-disk format) or Apache Arrow (in-memory columnar) is ideal. We aim to allow exporting a large layer or point set to Parquet so that Python/SQL engines can consume it. For example, if a user did a polyfill and wants the result as a table of cell IDs, Parquet is suitable. Or if we have a layer of values over many cells, Parquet can store columns: cell_id (or separate x,y,z columns) and value.

We will use Rust's Arrow or Parquet crate if available to implement an export. Perhaps provide a CLI command like `codex3d export-parquet -layer pop_density -out data.parquet`. Similarly for Arrow (which could also be used via Arrow's C data interface to feed into Python without disk intermediate).

The schema could be: - For cell data: columns = frame (u8), resolution (u8), x (i32), y(i32), z(i32), value (depending on type). - Or combine frame,resolution into one if consistent for dataset, and possibly combine x,y,z into a single cell_id 128-bit (Parquet does allow 128-bit ints). We might store cell_id as a fixed 16-byte binary in Parquet for compactness, and have separate columns for attributes.

- **GeoJSON (input and output):**

- Input: Accept polygons or paths from GeoJSON. For polygon, we'll parse it (likely via `geojson` crate or manual). Coordinates likely lat/lon – we need to handle if frame is geographic vs ECEF. If frame=0 is some planar coordinate, and input is lat/lon, we might need to project (maybe assume lat/lon WGS84 and transform to ECEF X,Y,Z for 3D). That could be an advanced feature: perhaps we define frame 1 = Earth WGS84 ECEF (where coordinates are in meters from Earth center). Then user can specify that frame or we detect GeoJSON with CRS. Initially, might support only planar coordinates or require the user to convert to appropriate coordinate system.

- Output: We can output results as GeoJSON for visualization:
    - For example, given a path (sequence of cell centers or perhaps edges), we can output a LineString in GeoJSON (with coordinates in whatever frame system, if Earth then lat/lon by converting back).
    - For a set of cells (polyfill result), we could output a MultiPoint of cell centers or ideally the actual cell boundaries as Polygons. The latter is challenging (truncated octahedron polygon has many vertices). We could output approximate shapes or bounding boxes for simplicity. Possibly skip boundaries for v1.
- We should ensure whatever output is clearly documented (units, coordinate frame).

It's likely easier to assume an Earth-centric use-case where we treat input lat/lon as something to convert to ECEF (Frame maybe "Earth (ECEF)"). That conversion we can incorporate if needed.

- **Other formats:**
- We might mention potential support for connecting to spatial databases (postGIS, etc.), but that can be future.
- Also mention maybe glTF or something for 3D visualization if we had time, but unnecessary now.

Synthetic Data Generators

To facilitate testing and demos, we will include utilities to generate synthetic spatial data: - **Random obstacle fields:** e.g., generate N random blocked cells in a region or some perlin noise field to simulate terrain difficulty. - **Geometric shapes:** e.g., a wall or a sphere of cells marked as obstacle. - **Path scenarios:** create a known maze or corridor structure to test pathfinding.

For instance, a CLI command `codex3d gen-random -region X -density 0.1 -flags blocked` could mark 10% of cells in region X as blocked randomly.

Or `codex3d gen-checkerboard -size 100` to alternate blocked/free in a pattern (to test diagonal navigation etc.).

We can also generate point datasets: `gen-points -distribution uniform -count M` to produce random points in a box, then test aggregation.

These generators will mainly be used in our examples and benchmarks, but they will also help users get started with how to populate the grid.

Developer Note: Keep generators simple but allow seeding for reproducibility.

Implementation Considerations

Finally, we outline some key considerations and best practices for implementing the above in Rust to ensure performance and maintainability:

Language Choice and Performance

We choose **Rust** for its combination of performance (comparable to C++), safety (to avoid segfaults, data races), and modern tooling. The system will be developed as a Rust library (`codex` crate perhaps) with the CLI as a separate binary target in the same repository.

- We will strive to make critical inner loops (neighbor calculation, A* expansions, etc.) allocation-free and amenable to inlining and optimization by the compiler.
- Use **iterators and slices** for safe high-level code, but drop down to unsafe or SIMD when profiling shows a bottleneck that can be optimized.

Rust also makes concurrency easier without data races – we'll exploit this for parallel tasks.

Concurrency and Parallelism

Many operations can be parallelized: - Aggregating a layer over many cells (we can split the cells among threads and then combine partial results). - Polyfilling a large region (split space or list of candidate cells). - Pathfinding is inherently sequential (graph search) and typically not parallelized, except maybe exploring two directions concurrently (bidirectional search already covers that conceptually). We won't parallelize A* itself (complex and usually not worth it), but we might parallelize running multiple path queries if needed.

We will use **Rayon** for high-level data parallelism. Rust's Rayon crate allows converting iterators to parallel iterators easily. For example, in polyfill, we can do something like:

```
cells.par_iter()
     .filter(|cell| cell_center_in_polygon(cell, poly))
     .collect()
```

This will distribute the filtering across threads automatically.

We should be mindful of thread overhead for small tasks, but for large ones, the speedup is huge.

SIMD: Some numeric operations can benefit from SIMD. Potential areas: - Distance calculations (computing many distances at once). - Possibly the neighbor offset addition can be vectorized (but 14 is not a power of 2, so maybe not trivially). - Aggregations like sum or min across an array can use horizontal SIMD operations. Rust stable now has SIMD in std for certain types (e.g., `std::arch::x86_64::_mm...` intrinsics or using the portable packed_simd crate). We won't prematurely optimize with SIMD until profiling indicates a bottleneck where it helps (like summing millions of values, etc.).

We can use `packed_simd` or `std::simd` (if stabilized by 2025 likely yes, possibly in nightly? There's `std::simd` as portable types). If stable, great, if not, we consider using it feature flagged.

One target might be speeding up point-in-polygon checks via SIMD (to check multiple points at once), but that might be overkill.

Testing (Proptest and Benchmarks)

We will write extensive tests for all components: - **Unit tests:** for basic functions (neighbor correctness, parent/child mapping, encoding/decoding, etc.). For example, verify that for a random cell, all 14 neighbors are indeed at distance equal to a face-sharing (we can check that their coordinate differences match allowed offsets). - **Property-based tests (using `proptest`):** These generate random scenarios to validate invariants. Examples: - Generate a random cell ID, compute parent then children, ensure the original cell is among those children (if we go up and down in resolution). - Generate two random cell IDs and run line traversal vs a brute-force check (like sample the line by very fine steps to see which cells it goes through, compare with our algorithm result). - Random point sets aggregated then roll-up and compare direct aggregation at coarse vs rolled up from fine (should match for sum, etc., small floating differences for mean). - Encoding/decoding roundtrips for random 128-bit patterns (ensuring that invalid combinations are either normalized or detected). - Pathfinding optimality: create a small grid with known weights (like a 3x3x3 area), enumerate all paths to confirm A* returns truly lowest cost.

Property tests help catch edge cases like off-by-one issues in coordinate halving, parity mismatches, etc.

- **Integration tests:** Use the CLI on some example input and verify the output format or counts (maybe via a test harness or just manual for now).

- **Benchmarks:** Using the Criterion crate, we will write benchmarks for key operations:

- Neighbor lookup speed (how many million neighbors/sec can we generate).
- A* solving a typical scenario (maybe a 100x100x10 region random blocks).
- Polyfill of a given shape size.
- Aggregation of a large dataset.

Criterion will give us timings and allow checking if changes regress performance.

We will aim to meet performance targets such as: - Neighbor lookup ~ nanoseconds level (since it's small math). - A* able to handle thousands of expansions per millisecond (should, if using a binary heap). - Polyfill thousands of cells per second for large shapes (maybe more; if millions of cells, measure accordingly).

Benchmarking early and often will guide optimizations (maybe we find that e.g. hashing of IDs is a bottleneck in maps – then we might use `FxHashMap` or different structure).

Versioning and Schema Evolution

As an open-source project, we'll follow semantic versioning for the library (e.g., v0.1.0 for initial release, moving to 1.0 when stable). The specification in this document corresponds to the initial version features.

ID Schema Versioning: If we ever need to change the bit layout of the ID or the meaning of fields, we must be careful. Because IDs may be stored persistently out in the world, a change would break things. To manage this: - Use the `frame` field or some reserved bits to indicate version. For example, we could designate frame 255 as a special value meaning "this ID uses an alternate layout". Or a top bit in the 8-bit resolution could flag a new format. - Alternatively, incorporate a version number in the Bech32m HRP (like prefix `cx3dv1` vs `cx3dv2`). - We will document the current format as v1. Future expansions should ideally fit into reserved bits or the extended 256-bit structure.

File/Data Schema: For outputs like CBOR/Parquet, we will include metadata: - CBOR could have a top-level like `{version:1, ...}`. - Parquet can have custom metadata key for version. - This way, if later we add columns or change units, we can handle backward compatibility.

Extensibility and Future Adapters

We design the system to be extensible: - The cost function trait allows plugging new behaviors easily. - The data layer design allows adding new layers or perhaps linking to external data sources (like a lazy layer that queries a database instead of memory). - The ID could potentially support other tilings (for example, one could imagine supporting the FCC lattice (rhombic dodecahedra) by a different frame or a flag – not now, but possible due to flexible bits). - The library API should be flexible enough to integrate with other systems. For example, one might want to use this grid as an index in a geospatial database or feed cell IDs to a vector database (like Qdrant) as metadata.

Specifically, we mention optional future adapters: - **Qdrant:** Qdrant is a vector similarity database. An adapter might store cell attributes or embeddings in Qdrant for similarity search. Not directly obvious for spatial grid, but maybe if each cell had a feature vector (like environmental signature), Qdrant could find similar cells. This is an out-of-scope idea, but knowing we might integrate, we keep our data structures serializable and accessible. - **ClickHouse:** A fast analytical database that supports large tables. We could store cell data in ClickHouse (which has good support for int128 as two uint64, or store as strings). We could allow writing layers to ClickHouse or even querying via a SQL generation. For now, Parquet export is a good step since ClickHouse can ingest Parquet. - **RocksDB / Key-Value stores:** For extremely large grids that cannot fit in RAM, storing data on disk by cell ID key might be needed. We could expose an interface to use a key-value store for layer data (e.g., a trait for layer storage). Then an implementation could use RocksDB or even Redis (for networked ephemeral store). - **Redis:** Real-time applications might share state via Redis. We could map cell IDs to values in Redis (like a distributed occupancy grid). Possibly an adapter that pushes updates to Redis channels or stores them with TTL.

While these are beyond initial implementation, our design will consider them: - Keep the core stateless or abstract over storage. - For example, define a `LayerStore` trait that has `get(cell)`, `set(cell, val)` so one can implement that trait for an in-memory HashMap or for an external DB. Then the algorithms (aggregation, pathfinding if reading cost from layers) can use the trait without caring about actual backend.

Extending to Surfaces or other grids: The spec is for 3D volume, but one could also use truncated octa grid for 2D surfaces (by constraining one dimension). We won't explicitly do that but it's worth noting that the core index could be adapted.

Finally, we will keep code modular: - Likely modules: `lattice` (for geometry, neighbor logic, coordinate transforms), `id` (for ID encode/decode), `layer` (data storage and aggregation), `path` (pathfinding algorithms), `io` (format import/export), `cli` (for CLI definitions). - We'll include documentation comments, examples, and possibly a mdBook or detailed README with usage examples.

Conclusion: This specification has detailed the design of the CODEX 3D spatial indexing and routing system. It covers the mathematical underpinnings (BCC lattice, truncated octahedral tiling) [5] [6], the hierarchical ID format and encoding [4], core operations (neighbors, rings, traversal), data handling and aggregation, routing algorithms (A*, hierarchical A*, D* Lite for replanning) [13], and I/O considerations, all with the aim of guiding a robust Rust implementation.

By following this spec, the engineering team should be able to implement a clean, correct, and maintainable system. The focus on multiscale routing and aggregation as first-class features will ensure CODEX stands out as a powerful tool for 3D spatial analysis. We will proceed with iterative development, validating each component against these requirements and ensuring thorough testing and documentation accompany the code.

Sources:

- Entezari et al., on BCC lattice properties (parity, Voronoi cell as truncated octahedron, 14 neighbors) [16] [5] [6]
- Wikipedia, *Truncated octahedron* (14 faces and space-filling nature) [1]
- Bitcoin bech32m documentation (for ID encoding rationale) [2] [3]
- Wikipedia, *D* (D* Lite algorithm usage for fast replanning) [13] [15]

---

[1] Truncated octahedron - Wikipedia
https://en.wikipedia.org/wiki/Truncated_octahedron

[2] Bech32(m) | Bitcoin Optech
https://bitcoinops.org/en/topics/bech32/

[3] A Deep Dive into Bech32 with Rust
https://mojoauth.com/binary-encoding-decoding/bech32-with-rust/

[4] [5] [6] [7] [8] [9] [16] eprints.cs.univie.ac.at
https://eprints.cs.univie.ac.at/5018/1/2004_-_Linear_and_cubic_box_splines.pdf

[10] rust-bitcoin/rust-bech32: Bech32 format encoding and decoding
https://github.com/rust-bitcoin/rust-bech32

[11] [12] Introduction to Acceleration Structures - Scratchapixel
https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-acceleration-structure/grid.html

[13] [14] [15] D* - Wikipedia
https://en.wikipedia.org/wiki/D*