Proposal: World-coord-keyed canonical encoding via a bidirectional UVW↔RGB atlas

Targeted at: nv-tlabs/lyra, Lyra 2.0 **Author:** MiLO + Opie (github.com/MiLO83/DownToEarth) **Status:** Architectural proposal — research / discussion, not a PR **License of this document:** MIT. Code references in this doc are from the Apache-2.0-licensed Lyra source tree.

In plain English (the wins, the losses, the gist)

The setup

Lyra 2's current way: keep a separate photo album for every photo it's ever taken of your house. If it photographs the kitchen from 5 angles, that's 5 album entries. To find "what's the kitchen wall?" it has to flip through every album looking for the right one.

Our proposed way: keep a single floor-plan map of the house. Every spot on the map has a unique address. No matter how many photos you take of the kitchen wall, it stays in one place on the map. To find it, glance at the map once.

That's the whole idea. Everything in the technical sections below follows from that.

Wins (why this could be worth it)

1. **No more memory bloat from looking at the same thing twice.** The camera can walk around a room and see the same wall fifty times. Old way: fifty new entries. New way: the wall's spot on the map gets refreshed in place. Memory stays the same size once the house is fully mapped.
2. **Finding stuff is one step, not fifty.** "Where's the kitchen wall?" Old way: flip through every photo album. New way: read the address off the map. The bigger the cache gets, the bigger the gap.
3. **Same place, same name.** The model never has to figure out "wait, is this kitchen wall from album 3 the same as kitchen wall from album 7?" The map address IS the identity. One thing, one name, no ambiguity.
4. **You can look at the map and read it.** Today the photo-album labels are codes like "album 47, photo 12, pixel (143, 89)." The new map's labels are actual map coordinates — you can literally print the map and a person can read it. Debugging stops being a guessing game.
5. **Neighbours stay neighbours.** On the map, two spots physically next to each other in the real world are also physically next to each other on the page. Your GPU loves this — it can grab a whole little chunk of map at once, instead of running off to fetch random photo albums from across storage. Modest speed-up, basically free.

Losses (why this isn't a slam dunk)

1. **The map has a grid; tiny details below grid size disappear.** At the cheapest setting (1 byte per axis), each map square is about 1 cm. Two things half a centimetre apart get squashed into the same square. For most architecture-scale stuff

this is fine. For very tiny detail you'd need a finer map (more bytes per axis), which costs more memory.

2. **You can't make a single map of the whole world.** A bedroom-sized map works great. A city-sized map needs to be folders of regional maps. A continent-sized map needs folders of folders. Lyra 2's specialty is "walk-forever" scenes — those need the folder-of-maps version, which is more engineering work. We sketched how but didn't build it.
3. **The model has to learn a new language.** Today's Lyra 2 has been trained to read photo-album labels. To switch to map addresses, the model needs to be re-taught — that's roughly six weeks of training on 64 of NVIDIA's most expensive GPUs. We can't do this; only NVIDIA can. So this is a "if you're already planning to retrain, consider folding this in" proposal, not a drop-in patch.
4. **The map doesn't remember WHEN you mapped each spot.** Photo albums have dates. "This album was from yesterday, that one was from last month — trust yesterday's more." The map just shows the latest state of each spot. To get the "when" back you'd attach a little timestamp byte to each map cell (we have a spare byte in our 4-byte-per-cell budget; it's already documented). Not lost, just an extra design choice.
5. **It only helps if there's stuff to map.** For a single static photo with no exploration, both approaches do basically the same thing. The wins kick in as the camera moves around and the same locations get re-observed — which IS Lyra 2's actual job, so it's the right shape, but worth being honest: a one-shot single-frame use-case sees almost no benefit.

The honest one-liner

> *Lyra 2 already computes the world-coords of every pixel — it just throws that information into a photo-album-style index. Re-indexing into a map-style atlas is the architectural shift; it's cleaner, faster, and bounded by scene size instead of frame count, but it costs a retrain and the addressable world is now finite (or hierarchical). That's the whole trade.*

Summary

Lyra 2's canonical-coord conditioning is **image-space-and-frame-indexed**: each pixel of the warped conditioning image encodes `(u_normalized, v_normalized, frame_slot_idx)`. The cross-attention then treats these as keys into a *frame-keyed* history of latents. This works, but two consequences fall out:

1. **Multiple past frames seeing the same 3D point produce different canonical-coord values.** Geometric correspondence is implicit and has to be re-learned at every attention layer.
2. **History size grows with time, not scene complexity.** A frame that re-observes already-known geometry still occupies a full slot.

Lyra 2's `Sparse3DCache` already computes per-pixel world coordinates (via `unproject_points` at `lyra2_model.py:2534`), so the world-coord information is present — it's simply not used as the canonical-coord key.

This proposal: **encode the canonical-coord image as quantized world coords (packed into RGB bytes via a bijective tile layout), and back it with a world-coord-keyed atlas instead of a frame-keyed cache.** Same warp machinery, different output encoding, different storage layout. Below: the mechanics, the wins, the trade-offs, and an estimated porting cost.

The current Lyra 2 mechanism (with citations)

I'll keep this brief — the paper covers it well — but anchor each claim to source.

Canonical-coord construction

`_build_canonical_spatial_coords` (`lyra2_model.py:1525-1545`):

```
xs = torch.linspace(-1.0, 1.0, W, device=device, dtype=dtype)
ys = torch.linspace(-1.0, 1.0, H, device=device, dtype=dtype)
yy, xx = torch.meshgrid(ys, xs, indexing="ij")
base_xy = torch.stack([xx, yy], dim=0)  # [2, H, W]
...
zs = torch.linspace(-1.0, 1.0, num_spatial_hist, device=device, dtype=dtype)
z = zs.view(num_spatial_hist, 1, 1, 1).expand(num_spatial_hist, 1, H, W)
coords = torch.cat([base_xy, z], dim=1)  # [N, 3, H, W]
```

So each pixel of slot `n` starts life as `(u_normalized, v_normalized, z_n)` where `z_n` is the slot's position along `[-1, 1]`. These get forward-warped via `forward_warp_multiframes` (`forward_warp_utils_pytorch.py:57`) through past camera poses + depths into the target view. The output of the warp is the canonical-coord image fed to the DiT as conditioning.

Storage and retrieval

`Sparse3DCache` (`lyra2_model.py:2488`) keeps:

```
self._world_points: list[torch.Tensor] = []  # each: [B, H', W', 3]
self._latent_indices: list[int] = []
self._frame_ids: list[int] = []
self._depths: list[torch.Tensor] = []
self._w2cs: list[torch.Tensor] = []
self._Ks: list[torch.Tensor] = []
self._rgbs: dict[int, torch.Tensor] = {}
```

`add()` calls `unproject_points` to compute world coords per pixel (line 2534), then stores those alongside the depth/w2c/K and the RGB. **The world coords are computed and stored.** Retrieval (`Sparse3DCache.retrieve`) selects past frames by visibility overlap with the target view and returns their latent indices for cross-attention.

So: **frame-keyed cache, image-space-plus-frame canonical-coords.**

The proposed alternative

Encoding change

Replace the `(u_norm, v_norm, frame_slot)` encoding with **quantized world coords packed into RGB bytes**:

```
# Quantize world coords to [0, 2^B - 1] per axis (B = bits per channel).
# For B = 8 (RGBA8): 256 levels per axis = 16.7M unique 3D positions in a
# bounded world. For B = 16 (RGBA16UI): 65,536 levels = 281T positions.
def world_to_canonical_rgb(world_xyz: Tensor, world_min: Tensor,
                           world_max: Tensor, bits: int = 8) -> Tensor:
    n = (1 << bits) - 1
    normed = (world_xyz - world_min) / (world_max - world_min)  # [0, 1]
    return (normed.clamp(0, 1) * n).to(torch.uint8 if bits == 8 else torch.uint16)
```

This is byte-perfect bijective: the bytes in the canonical image are the world coords (modulo a known world_min/max transform). The current warp machinery transports them correctly — `forward_warp_multiframes` doesn't care whether `frame1` is RGB, normalized image coords, or quantized world coords; it just resamples per the depth + camera transform.

Storage change

Replace the frame-keyed `Sparse3DCache` with a **world-coord-keyed atlas** addressed via a 2D tile layout (the bijection: $256^3 \to 4096^2$ for a single- texture-array fit, hierarchical sparse beyond that):

```
# Pack (u, v, w) ∈ [0, 256)³ into a 2D atlas position via a 16×16 tile grid.
# Each tile is one full w-slice of size 256×256.
def voxel_to_atlas(u: int, v: int, w: int) -> tuple[int, int]:
    return ((w & 15) << 8) | u, ((w >> 4) << 8) | v
```

The atlas stores whatever Lyra 2 currently stores per past frame — latent feature vectors, RGB, or both — but indexed by 3D position instead of by `(frame_id, pixel_position)`. When the same 3D point gets observed by multiple frames, the new write **strengthens or replaces** the existing slot rather than adding a new frame entry.

This bijection is documented + verified in `voxgaussian/pipeline/uvw_atlas.py` with an exhaustive 16.7M-pair round-trip test. The identity mapping (atlas-position ↔ canonical-pass RGB) is *mathematically inherited* and costs zero VRAM — only the payload uses storage.

What changes in the cross-attention

Currently cross-attention is over frame latents, keyed by the warped `(u_norm, v_norm, frame_slot)`. With the proposal, cross-attention is over the atlas, keyed by the warped `(world_x, world_y, world_z)`. The attention math is unchanged; only the storage layout changes.

Why this is interesting (the wins)

1. **Geometric correspondence becomes structural, not learned.** A 3D point seen by frames 3, 7, and 12 has *one* key — its world coord — instead of three different `(u, v, frame_slot)` triples that the model must learn to treat as equivalent. This should reduce the inductive load on the cross-attention.
2. **Memory scales with scene complexity, not time.** Re-observing known geometry reinforces existing atlas slots; it doesn't allocate new frame slots. For long camera trajectories that revisit already- explored regions (which Lyra 2's 90 m worlds do, by design), this bounds memory growth.
3. **O(1) "is this 3D point known?" lookup.** Currently `Sparse3DCache.retrieve` scores all candidate frames by visibility overlap (see `lyra2_model.py:2737–2832`). With a world-coord atlas, the question becomes a texture sample at the warped coord — single instruction.
4. **Spatial locality on GPU.** A tile-layout (or Hilbert-curve) bijection preserves 3D-neighbour adjacency in the 2D atlas, so cache prefetch helps for the typical "raymarch-y" access pattern.
5. **Debuggable conditioning.** Visualizing the canonical-coord image shows you exactly what the model sees: pixel colors *are* coords. No inverse-projection step to interpret it.

Trade-offs (the catches)

1. **Bounded vs. unbounded scenes.** 256^3 fits in 4096^2 (16.7 M atlas slots, ≈ 2.5 m world at 1 cm cells). Lyra 2's 90 m+ worlds need a hierarchical sparse-tiled version — atlas grows octree-style as the camera moves into new regions. Doable but more engineering. RGBA16 pushes the bounded reach to ≈ 655 m at 1 cm cells.
2. **Retraining required.** The model has learned to read `(u, v, frame_slot)`. Adopting world-coord encoding changes its inputs — at minimum the conditioning head needs fine-tuning, more likely full retraining. Cost-prohibitive to attempt this without NVIDIA's training infrastructure.
3. **Loss of temporal information.** Frame index carries "when did we see this" signal that can be useful for weighting recent observations higher (drift correction). To preserve this in the atlas-keyed scheme, attach a `last_updated_iter` byte to each atlas slot — uses one of the 4 RGBA bytes the address doesn't need for spatial identity.
4. **Quantization error.** World coords get bucketed into the atlas resolution. At 8-bit per axis (256^3), a 2.5 m world has 1 cm cells — probably fine for the kind of geometry Lyra 2 works with, but noticeable for very small detail. RGBA16 fixes this entirely.
5. **Doesn't help the unbounded-walk case directly.** Lyra 2's spec is *autoregressive long-horizon generation*. For each new chunk, the camera is somewhere new and we need to extend the world. A pure bounded atlas saturates; a hierarchical one is required for the full spec to be preserved.

Performance cross-reference (before vs. after)

All numbers calibrated to Lyra 2's documented config: **832 × 480 resolution, 80 frames per chunk, num_spatial_hist = 5, Sparse3DCache downsample = 4**. ✶ = calculated from the codebase. † = estimated. ‡ = qualitative.

Dimension	Current (Lyra 2 frame-keyed)	Proposed (world-coord atlas)	Delta
Canonical-coord image size, per chunk	5 × 3 × 480 × 832 × fp16 = **11.4 MB** ✶	RGBA8: **5.71 MB** / RGBA16UI: 11.4 MB / RGBA32F: 22.9 MB ✶	RGBA8: **2× smaller** than fp16 / 4× smaller than fp32; RGBA16: parity
Cache memory: `_world_points`, after 80 frames	80 × (480/4) × (832/4) × 3 × fp32 = **24 MB** ✶	256^3 atlas × 4 B (RGBA8) = **64 MB** fixed (1024^3 → ~4 GB texture array) †	Larger up front, **flat with time** vs. linear
Cache memory: with `store_values=True` (full depth + w2c + K + RGB latents)	80 × (480 × 832 × fp32 depth + intrinsics + latents) ≈ **130 MB + frame latents** ✶	Atlas + per-slot payload bytes; no per-frame redundant storage. ~64 MB to ~1 GB scene-dependent. †	Scene-complexity-bound, not time-bound
Memory growth on revisits (camera re-enters region already seen)	Linear: every frame adds a slot whether new or not	Flat: same world-coord slot gets reinforced, no new allocation	**O(time) → O(scene)**

Dimension	Current (Lyra 2 frame-keyed)	Proposed (world-coord atlas)	Delta
`Sparse3DCache.retrieve()` **cost per target view**	Scores all N candidate slots by visibility overlap; iterate + sort	Single 4-instruction warp + texture sample per target pixel	**O(N) → O(1)** per pixel
Cross-attention KV count	num_spatial_hist (5) × tokens-per-frame	num_spatial_hist (5) × tokens-per-frame (unchanged — same model arch)	Same
Distinct keys per 3D point across views	One per past-frame-that-saw-it; the model must learn equivalence	**Exactly one** (the quantized world coord)	Inductive load ↓
Cache write atomicity (multi-view fusion)	Sequential per-frame appends; merge logic in `retrieve()`	Atomic increment / max into atlas slots; GPU `atomicAdd` on `R32_UINT`	Trivially parallel
GPU cache locality (spatial neighbours)	Random — frames stored independently; access pattern depends on the warp permutation	Preserved — 16×16 tile layout keeps 3D neighbours within ~256 px in the 2D atlas	Texture-cache friendly
Quantization error	None (fp32 world coords retained internally)	**1 LSB per axis**: 1 cm at 256^3/2.5 m world (RGBA8), <0.1 mm at RGBA16. None at RGBA32 ✶	RGBA8: low / RGBA16: negligible
Disocclusion-hole rendering quality	Cleanly black where no past frame saw the pixel	Same — atlas lookup returns (0,0,0) where no slot has been written	Identical
Long-horizon (Lyra 2 spec, 90 m walk-through)	Frame count grows linearly with trajectory length; retrieval slows	**Requires hierarchical sparse atlas** — bounded variant saturates beyond world_max ‡	Open engineering question, advantage *if* sparse-tiled is built
Conditioning interpretability (visualize the canonical image)	Hard to read (normalized image-coords + frame idx)	RGB literally encodes (u, v, w) — visual debug is the bytes ‡	Strict win for development
Code surface area (LoC change)	—	`_build_canonical_spatial_coords` rewrite ~~(50 LoC) +~~ ~~`Sparse3DCache`~~ ~~dual-key mode~~ (100 LoC) + new `uvw_atlas.py` ~~(150 LoC).~~ ~~**~~300 LoC net add.** †	Small
Training-compat impact	—	**Full retrain or fine-tune the conditioning head**: model expects different statistics on the	Significant — needs NVIDIA

Dimension	Current (Lyra 2, frame-keyed)	Proposed (world-coord atlas)	Delta
		input channels. ‡	training infra
Inference latency (per 80-frame step)	~194 s on GB200 (full), ~15 s (DMD-distilled) ✶ (from paper)	Estimate: ~1–3 % reduction from cheaper canonical-coord pass + O(1) retrieval; possible 5–10 % reduction at long-horizon revisits where current retrieve() dominates ‡	Modest, possibly meaningful at scale

Byte-length comparison (per-channel precision)

Held constant: Lyra 2 res (832 × 480), 5 spatial slots, 3 coord channels, a single inference chunk. Atlas dims assume the maximum single-texture limit (16384^2) at each format; beyond that you need a texture array or sparse-tiled scheme.

Format	Bytes / axis	Max axis val	World @ 1 cm cells	World @ 1 mm cells	Canonical img (5×3×480×832)	Single-texture atlas footprint	Identity flavour	GPU support
RGBA8 (uint8)	1	256	2.56 m	25.6 cm	**5.71 MB**	16384^2 × 4 B = **1 GB**	byte = coord (exact)	universal
RGBA16UI (uint16)	2	65,536	**655 m**	65.5 m	11.4 MB	16384^2 × 8 B = **2 GB**	uint16 = coord (exact)	WebGL2 + `EXT_color_buffer_integer`
RGBA16F (half)	2	1,024 (mantissa)	10.2 m	1.02 m	11.4 MB	16384^2 × 8 B = **2 GB**	float ≈ coord (exact ≤ 2^{10}, lossy after)	WebGL2 native
RGBA32F (float32)	4	16.7M (exact)	**167 km**	**16.7 km**	22.9 MB	16384^2 × 16 B = **4 GB**	float = coord (exact ≤ 2^{24})	WebGL2 native
RGBA32UI (uint32)	4	4.29B	**42,000 km**	**4,200 km**	22.9 MB	16384^2 × 16 B = **4 GB**	uint32 = coord (exact)	WebGL2 + `EXT_color_buffer_integer`

Same dimensions vs. Lyra 2's existing fp16 / fp32 canonical-coord image:

Lyra 2 today	Proposed equivalent	Image size delta	Identity delta
Current: fp16 (2 B/axis)	RGBA8 (1 B/axis)	**2× smaller image**	Quantize: 1 cm @ 2.5 m world, but **byte-perfect** vs. fp16's float-rounding

Lyra 2 today	Proposed equivalent	Image size delta	Identity delta
Current: fp16 (2 B/axis)	RGBA16UI (2 B/axis)	**Same size**	byte-perfect vs. float-rounding; 655 m @ 1 cm world available
Current: fp32 (4 B/axis)	RGBA16UI (2 B/axis)	**2× smaller**	byte-perfect; 655 m @ 1 cm reach
Current: fp32 (4 B/axis)	RGBA32UI (4 B/axis)	Same size	byte-perfect vs. float-rounding; effectively unbounded

Recommendation by Lyra 2 use-case

Scene scale	Recommended format	Atlas storage	Notes
≤ 2.5 m (single room / character)	**RGBA8**	64 MB at 256^3	Plenty for the bounded variant; pairs with single 4096^2 texture, the original DownToEarth target
2.5–655 m (city block, building cluster)	**RGBA16UI**	1–2 GB texture array	Matches Lyra 2's typical 90 m world spec at sub-cm precision
655 m – 167 km (urban / regional)	**RGBA32F**	Sparse-tiled, scene-dependent	Mantissa exact to 2^{24}; lose strict byte-identity outside that
Unbounded (continent+)	**Hierarchical sparse + RGBA32UI**	Octree-style page table	Byte-identity preserved **within each leaf tile**; cross-tile is one indirection

The format choice is **independent** of the architectural proposal — even at RGBA8 you get the structural wins (one key per 3D point, O(1) retrieval, memory bounded by scene-not-time). The byte width just sets the addressable world size.

What this table is *not*

- **Not a quality-score comparison.** SSIM, LPIPS, FID — none of these can be predicted from the architecture change alone. They require a retrain to measure. The above is purely the *computational* and *memory* analysis.
- **Not a benchmark.** Numbers above are calculated or estimated from the codebase + the paper. No actual runs were performed (the proposers don't have GB200s).
- **Not a free win.** The retraining cost is the elephant in the room. The argument is the per-inference math is cleaner and the memory growth on long trajectories is bounded — *if* a retrain is in the budget anyway, this is the kind of architectural shift to fold in.

Concrete porting sketch (if anyone wants to try)

Three files change, roughly:

1. `lyra_2/_src/models/lyra2_model.py::_build_canonical_spatial_coords`

```
def _build_canonical_spatial_coords(H, W, num_spatial_hist, device, dtype, *,
                                    world_min, world_max, bits=8):
```

```
    # Generate a per-pixel world-coord initialization that the warp will
    # then transport. Each spatial slot still distinguishes via the 3rd
    # axis — but the axis is "depth tag" not "frame slot" so warps from
    # different past frames that landed on the same 3D point produce the
    # same canonical RGB.
    xs = torch.linspace(world_min[0], world_max[0], W, ...)
    ys = torch.linspace(world_min[1], world_max[1], H, ...)
    ...
    return coords  # [N, 3, H, W] in WORLD units (will be re-quantized post-warp)
```

After `forward_warp_multiframes` runs, quantize the warped output to `uint8 / uint16` via `world_to_canonical_rgb`. The downstream `_pixelshuffle_hw_to_latent` (line 1568) is dtype-agnostic.

2. `lyra_2/_src/models/lyra2_model.py::Sparse3DCache`

Add a world-coord-keyed mode alongside the existing frame-keyed one. A single `dict[atlas_pos, payload]` (or a sparse tensor) keyed by the 2D atlas position. Existing `add()` writes to *both* (frame-keyed *and* atlas-keyed) during a transition period; `retrieve()` gains an `use_atlas: bool` flag.

3. New `lyra_2/_src/datasets/uvw_atlas.py`

The bijection helpers. Three functions:

```
def world_to_atlas(world_xyz, world_min, world_max, bits=8):
    """World 3-vec → (atlas_x, atlas_y) 2-vec."""

def atlas_to_world(atlas_xy, world_min, world_max, bits=8):
    """Inverse: (atlas_x, atlas_y) → world 3-vec."""

def quantize_world_to_rgb(world_xyz, world_min, world_max, bits=8):
    """World 3-vec → packed (R, G, B) byte triple. Color IS coord."""
```

All pure-arithmetic, no learnable parameters. We've open-sourced this exact module (MIT) at `voxgaussian/pipeline/uvw_atlas.py` — Lyra-friendly to copy/adapt under Apache-2.0 compatibility.

Bonus: a same-resolution 1-bit occupancy bitmap

Independent of the byte-width family above, a parallel **1-bit-per-voxel companion texture** at the same grid resolution is a low-cost addition worth flagging. It answers a single question — "*is this slot populated?*" — and lets the cross-attention / inference shader **early-skip the multi-byte main-atlas read** for empty voxels.

Cost: 2 MB at 256^3 (24× smaller than 3-byte-per-voxel RGB storage, 32× smaller than RGBA8). At 1024^3 it's 134 MB.

Three compounding wins:

1. **Storage** — 24× less than 3-byte/voxel
2. **Bandwidth** — same factor; for bandwidth-bound shaders, 10-20× faster scans
3. **Cache locality** — 24× more voxels fit in L1/L2 GPU cache

Pairing with sparse RGB storage (only allocate bytes for populated voxels): dense 1-bit mask gives O(1) addressability + near-hash-table memory cost. For Lyra 2's typical scenes (~5% surface occupancy), the total memory for occupancy + RGB

data drops from 50 MB → ~4.6 MB at 256^3.

For Lyra 2 specifically: the canonical-coord image has implicit emptiness today (pixels where the warp didn't write content are zeros), but it's a 3-channel zero check — `all(rgb == 0)` — which the cross-attention has to learn to interpret. An explicit 1-bit occupancy channel would:

- Make "is this voxel real or padding?" question structural, not learned
- Cost ~50 KB per inference chunk at Lyra 2's 832 × 480 resolution
- No model retraining needed if exposed as a separate ControlNet input

GLSL is one texel fetch + a shift + an AND:

```
uniform usampler2D occupancyBitmap;   // R8UI, dims atlasW/8 × atlasH
bool is_occupied(ivec2 atlas_xy) {
    uint byte_v = texelFetch(occupancyBitmap,
                             ivec2(atlas_xy.x >> 3, atlas_xy.y), 0).r;
    return ((byte_v >> (atlas_xy.x & 7)) & 1u) != 0u;
}
```

Eight voxels packed per byte along the X-axis preserves cache locality for "scan a row of voxels" access patterns.

Implementation reference — the `OccupancyBitmap` class in `voxgaussian/pipeline/uvw_atlas.py` ships this exact pattern at the DownToEarth scale (256^3). MIT-licensed, doctested, exhaustive round-trip self-test passing (`python -m pipeline.uvw_atlas` → "OK"). 2 MB for the 4096^2 atlas, 32× smaller than the RGBA8 summary atlas it pairs with. Lyra-2-friendly to adapt under Apache-2.0 compatibility.

Extension: 2-bit-per-voxel for demand-driven sparse streaming

A natural extension of the 1-bit occupancy mask: **double it to 2 bits per voxel** so each voxel slot carries occupancy AND a per-frame "ray-touched" bit. Storage doubles to 4 MB at 4096^2 (still trivial), and the runtime gains a precise log of which voxels actually contributed to a pixel that frame.

Crucially, the render-flag bit lives at the same atlas address as the voxel's RGBA data, keyed by the UVW bijection from §3. The runtime raymarcher computes one `voxel_to_atlas(u, v, w)` lookup per ray hit; that single atlas coordinate then serves both reads (sample the voxel's RGBA from the main atlas) and writes (set the render-flag bit in the occupancy/touched bitmap via `imageAtomicOr`). The two bitmaps share an address space because **the byte triple that names the voxel is the same triple that addresses both its data and its flag**. This is the runtime payoff of the bijection: one lookup, two uses, zero indirection.

Bit layout per byte (4 voxels packed):

```
voxel n at byte (n>>2), bits ((n&3)*2)  occupancy
                             ((n&3)*2+1) touched-this-frame
```

The touched bit is **set by the raymarcher** via `imageAtomicOr` during the per-pixel ray traversal, **OR-reduced to chunk granularity** at end-of-frame (returns the set of chunks the renderer actually visited), and **cleared once per frame** via a masked AND that preserves occupancy:

```
uniform usampler2D voxelState;            // R8UI, 2-bit-per-voxel layout
layout(r8ui) uniform uimage2D voxelStateImg;  // for atomic-or

bool is_occupied(ivec3 voxel_xyz) {
    ivec2 a = voxel_to_atlas(voxel_xyz);
    uint b = texelFetch(voxelState, ivec2(a.x >> 2, a.y), 0).r;
```

```
    return ((b >> ((a.x & 3) * 2)) & 1u) == 1u;
}

void mark_touched(ivec3 voxel_xyz) {
    ivec2 a = voxel_to_atlas(voxel_xyz);
    uint bit = 1u << ((a.x & 3) * 2 + 1);
    imageAtomicOr(voxelStateImg, ivec2(a.x >> 2, a.y), bit);
}
```

← SCENE DEMO UVW EXPLAINER SOURCE ON GITHUB

CPU-side, the **same** `OccupancyBitmap` **class** handles both modes via a `bits_per_voxel ∈ {1, 2}` constructor argument and exposes `set_touched`, `get_touched`, `clear_touched`, `count_touched`, and `touched_chunks(chunk_size=16)` which OR-reduces to `{(cx, cy, cz)}` chunk keys. The reduction is the **streaming signal**: any chunk in that set must remain VRAM-resident; chunks outside it are eviction candidates (typically with N-frame hysteresis).

Demand-driven sparse voxel streaming

The 2-bit bitmap is the on-GPU half of an **UE5-Nanite-style demand-driven streaming pipeline applied to voxels**: the renderer announces what it needed via the touched bits, the streamer keeps those chunks resident, evicts the rest.

Why this is the correct architecture for omnidirectional / VR playback: a frustum-based culler would pre-load only what the camera currently faces. In VR or 360° projections, the user's head rotates faster than chunks can stream from disk → pop-in, black frames. The touched-bitmap approach makes no view-direction assumption — whatever rays were cast (per-eye frustums for VR, equirectangular sphere for 360°, planar for FPV) populate `touched_chunks()` correctly. The streamer responds to actual demand.

Per-frame loop:

```
clear_touched()                 # ~10 µs for 4 MB
render_frame()                  # rays atomic-or their touched bits
touched = touched_chunks()      # OR-reduce to chunk keys
streamer.update(touched)        # queue SSD fetches; mark cold chunks
```

Prior art for the demand-driven pattern:

- John Carmack's MegaTexture / id Tech 5 Virtual Texturing (2D textures)
- Unreal Engine 5 Nanite (triangle clusters)
- NVIDIA GVDB (sparse voxel pages)
- Atomontage / John Lin's voxel engines (per-pixel voxel paging)

The voxel application is novel in its combination with the UVW bijection (each voxel address is a fixed byte triple, so the touched bitmap is the same memory layout as the atlas itself), but the demand-driven pattern itself is well-established.

Implementation status: shipped in PR #61 as of 2026-05-20 across two commits — `38907b3` adds the 2-bit OccupancyBitmap mode (occupancy

- render-flag, masked-AND clear, OR-reduce to chunk granularity) plus the CPU-first model_loader path that makes int8 Lyra 2 inference actually fit on 16 GB GPUs; `05670ec` adds first-class voxel-coord helpers (`set_by_voxel`, `set_touched_by_voxel`, etc.) so the API explicitly uses the UVW bijection rather than raw atlas indices. The shader-side `imageAtomicOr` wiring is in scope for future PRs once a runtime voxel renderer is integrated.

Why this matters: TB-scale worlds on 16 GB VRAM

The UVW bijection (§3-4) + occupancy bitmap (§6.5) + demand-driven streaming (§6.5.2) are independently described above, but their combined consequence deserves direct emphasis: **arbitrarily-large voxel worlds become renderable on consumer hardware**, where "arbitrarily-large" is bounded only by available disk and not by GPU memory.

The scale problem

At 1 cm voxel resolution, voxel counts grow with world volume:

World envelope	Voxel count	RGBA8 atlas	RGBA16 atlas
Bedroom (3 m^3)	~3×10^6	12 MB	24 MB
Apartment (200 m^3)	~2×10^9	8 GB	16 GB
Small island (5×10^7 m^3 = 1 km^2 × 50 m)	**5×10^{10}**	**200 GB**	400 GB
Town (10 km^2 × 80 m)	8×10^{11}	3.2 TB	6.4 TB
City (100 km^2 × 100 m)	10^{13}	40 TB	80 TB
Continental tile (Microsoft Flight Sim Earth equivalent)	~10^{15}	~2.5 PB	5 PB

Past the "apartment" row, no consumer-class GPU can hold the full atlas. Past the "island" row, no single SSD can hold it either; the larger scales require cloud-tier object storage with local sliding-window caching.

The decoupling: working set vs total set

For a screen of N pixels, **at most N voxels contribute to a frame** (one per pixel terminator), plus the K voxels each ray traverses through empty space before terminating (DDA step count, typically K ≈ 5-20 for sparse scenes with the occupancy bitmap accelerating skips).

For 1080p (2.1 M pixels) at 1 cm voxel: ~10-40 million voxels touched per frame. At 16 bytes per voxel (RGBA8 + occupancy + touched + LOD metadata), the *visible-this-frame footprint* is **160-640 MB**, regardless of total world size.

This is the fundamental dimensional argument: **the per-frame VRAM requirement scales with screen resolution, not world size**. A 5 TB world and a 50 GB world both fit in the same ~6 GB VRAM working set for the same screen.

Memory hierarchy budgets

For a Lyra-2-Lite-baked 1 km^2 island (200 GB total RGBA8 atlas with 2-bit occupancy/touched bitmap = 4 MB chunk-cull layer + 200 GB detail), the per-tier breakdown on an RTX 5060 Ti (16 GB) host:

Tier	Capacity	Contents	Access latency
GPU L1 / registers	~1 MB	current ray's neighbourhood	<1 ns
GPU L2 (Blackwell)	~32 MB	recently-accessed chunk neighbourhood	~10 ns
VRAM working set	**~6 GB** of 16 GB	**~200 16^3 chunks** of detail near camera + always-resident LOD pyramid	~200 ns
System RAM (page cache)	~16 GB	~1 GB of warm chunks staging for VRAM upload	~100 µs
NVMe SSD	1-2 TB	full atlas (200 GB) + neighbouring scenes	~50 µs random read

Tier	Capacity	Content	Access latency
Cloud (if applicable)	unbounded	tiles for unvisited regions	~10-50 ms

Each tier holds **only what the tier below it has touched recently** plus speculative pre-fetch. The 2-bit touched bitmap is the signal that drives every promotion/eviction decision.

Per-frame bandwidth math

At 60 fps with the player moving at 5 m/s (typical FPS camera speed):

```
Camera advance per frame:        5 m/s ÷ 60 fps = 8.3 cm
New voxels in view direction:    8.3 voxels at 1 cm = ~8 voxels deep
New chunks per frame:            ~0.5 (statistical) at 16³ chunk size
NVMe read per frame:             ~16 KB/frame (one 16³ × RGBA8 chunk)
                                  ≈ 1 MB/sec sustained
NVMe Gen4 capability:            5–7 GB/sec
Headroom:                        ~5000× under-utilisation
```

For typical movement, NVMe is **enormously over-provisioned**. The limiting factor is not bandwidth but **latency to first useful pixel** when entering a new region (cold cache).

For worst-case motion (teleport, instantaneous position change):

```
Worst-case new chunks:           ~500 (typical visible chunk count at 16³)
Worst-case NVMe load:            500 × 16 KB = 8 MB
NVMe Gen4 transfer time:         ~1.6 ms
                                  → fits in single 16.67 ms frame budget
```

A teleport completes within one frame's wall budget on Gen4 NVMe. Brief cold-cache LOD fallback handles the transition gracefully.

LOD pyramid: scaling past bandwidth limits

For very large worlds (>10 km^2) or extreme motion (>100 m/s), bandwidth eventually becomes the bottleneck. The standard mitigation is a resolution pyramid keyed by camera distance:

LOD level	Voxel size	Storage at 1 km^2	Distance range
0	1 cm	200 GB	0-30 m
1	2 cm	25 GB	30-60 m
2	4 cm	3 GB	60-150 m
3	8 cm	400 MB	150-300 m
4	16 cm	50 MB	300-600 m
5	32 cm	6 MB	600+ m (always resident)

LOD-5 is small enough to be permanent in VRAM, ensuring **every pixel always resolves to something visible** even on cold cache. Higher LODs stream in as the camera approaches. The same UVW atlas format works at every level; only the world-to-voxel scale changes.

With LOD active, the working set is bounded by visible detail at all distances simultaneously, not by world size. A 100 km² world and a 1 km² world both fit in ~6 GB working set.

Where this breaks

Five hard limits worth documenting for completeness:

1. **Cold-teleport quality dip** — first frame after instant position change is LOD-5 only at close range until ~1 frame later when LOD-0 chunks land. ~16 ms visible "blur" during teleport.
2. **Sustained extreme motion** (>100 m/s) — NVMe bandwidth becomes limiting for the highest-detail layer. Mitigation: increase always-resident LOD-1 layer, or accept transient detail loss during fast traversal.
3. **Anything past 720°/sec head rotation** — beyond the typical 500°/sec human max. Not a concern for normal use.
4. **Dynamic content** — moving characters/particles/destruction not addressed by this architecture. Requires a separate dynamic layer (small per-object voxel atlases with transforms, composited against the static raymarch). Out of scope for this proposal.
5. **Storage beyond local SSD** — for worlds >2 TB, requires network-tier object storage with the local SSD acting as a regional cache. Achievable (Flight Simulator does this at 2.5 PB scale) but adds a network latency layer that's > NVMe latency by ~100×.

Prior art and novelty

The pattern (render-driven streaming of paged data) is well-established:

System	Year	Paged unit
id Tech 5 MegaTexture (Rage)	2011	2D texture pages
id Tech 6 Virtual Texturing	2016	Generalised
NVIDIA GVDB	2018	Sparse voxel pages
Unreal Engine 5 Nanite	2022	Triangle cluster pages
Microsoft Flight Simulator 2024	2024	Regional terrain tiles, cloud-cached

The novelty in this proposal isn't the streaming pattern itself. It's the **integration with the UVW bijection**: the chunk's *address-key* IS its rendering color channel (no separate index lookup), the *occupancy bitmap* is the same memory layout as the atlas (one texture-fetch tests both occupancy and touched), and the *world generator* (Lyra 2 / Lyra 2 Lite) emits the same atlas format the renderer consumes (no post-bake conversion).

The combination yields a **content-creation + rendering pipeline that is internally consistent** from the diffusion bake to the per-pixel ray. The streaming voxel renderer is not bolted onto an unrelated generator; the generator's output format is the renderer's input format.

Future architectural directions

The proposal above is grounded in what we can demonstrate today against Lyra 2's current architecture. This section captures two forward-looking architectural ideas that emerged from our design conversations and that we think are worth

recording on the public record — not because they're in the patch (they aren't), but because they're a natural continuation of the UVW-conditioning thread and may inform Lyra 3 / sibling-model design.

These are intuitions, not validated results. We can't run the experiments. We're recording them publicly now so that (a) the ideas have a timestamp and provenance, (b) anyone considering similar architectures has prior context, and (c) the team can engage if any of these are interesting enough to discuss.

A. Two-pass structure-then-channels diffusion

Standard diffusion (DDPM, EDM, Wan 2.1, Lyra 2) uses a single denoiser network that handles all noise levels via timestep conditioning. The network implicitly learns to solve gross structure at high noise levels and fine pixel detail at low noise levels — but the architectural separation is implicit, every parameter is involved in every step.

The proposal: make this explicit by splitting diffusion into two architectural stages:

1. **Structure pass (few large-noise steps):** A specialised denoiser that consumes ONLY the world-coord canonical channel (the UVW bijection from §4.1: 3 bytes per pixel encoding voxel position). It outputs a coord map: "wall at (50, 70, 40), sky at (0, 200, 0)." No RGB. Roughly 4 noise steps.
2. **Channel pass (few small-noise steps):** A second, smaller denoiser that takes the locked coord map as conditioning and resolves RGB / lighting per pixel. Because the structure is fixed, the conditional distribution over RGB given UVW is much narrower than the unconditional distribution — the second-stage problem is dramatically easier. Roughly 2-4 noise steps.

Adjacent literature: the closest published expression is **eDiff-I** (NVIDIA Research, 2022) which uses expert denoisers per noise range. This proposal differs by splitting on *representational role* (UVW position vs RGB channels) rather than noise magnitude. Cascaded Diffusion (Imagen) does a related thing across resolution stages.

Why it pairs with our UVW work: the structure pass operates *directly* on the world-coord canonical channel. The bijection from §3 becomes the central data type, not a retrofitted side channel. The model is architecturally UVW-native rather than UVW-adapted.

Estimated compute saving: stage 1 has 1-3 channels of input vs Lyra 2's RGB+conditioning stack, smaller model viable (~~500M params would likely suffice for the structure pass). Stage 2 is colourisation-given-structure, also smaller~~ (200M params). Total inference cost potentially **3-5× lower than monolithic 14B Lyra 2** at similar quality.

Open question: does training the two stages independently and composing them at inference time match the quality of jointly training one large model? eDiff-I's results suggest yes within reason; the answer for video diffusion is unverified.

B. Progressive bit-depth ladder with bounded refinement

A finer-grained generalisation of A: instead of one monolithic diffusion that decides everything at full RGB precision, decompose into a *ladder* of stages each operating at a higher bit-depth than the last, with each stage bounded to small deviations from the previous stage's output.

The decomposition is per-channel, not luma-then-chroma. RGB is three independent intensity channels (3× the same byte-spectrum), so the ladder applies independently and identically to red, green, and blue. There's no luma extraction, no chroma stage — every rung is full RGB at progressively-higher bit-depth.

The proposal:

Stage (rung)	Per channel	Total colours	Per-pixel search space	Allowed deviation from previous
1 (1 BPP)	2 levels each	8 (2^3)	8 colours per pixel	free

Stage (rung)	Per channel	Total colours	Per-pixel search space	Allowed deviation from previous
2 (2 BPP)	4 levels each	64 (4^3)	±1 per channel = 27 deltas	±1 channel-level
3 (3 BPP)	8 levels each	512 (8^3)	±1 per channel = 27 deltas	±1
4 (4 BPP)	16 levels each	4 k (16^3)	±1 per channel = 27 deltas	±1
5 (5 BPP)	32 levels each	33 k (32^3)	±1 per channel = 27 deltas	±1
6 (6 BPP)	64 levels each	262 k (64^3)	±1 per channel = 27 deltas	±1
7 (7 BPP)	128 levels each	2.1 M (128^3)	±1 per channel = 27 deltas	±1
8 (8 BPP)	256 levels each	16.7 M (256^3)	±1 per channel = 27 deltas	±1 (final RGB)

Reverse-posterising as generation. Stage 1 generates a heavily-posterised RGB image (8 colours total — pure black/white/red/green/blue/etc, a "paint by numbers" stencil). Each subsequent stage doubles the per-channel level count, and the model picks one of 27 deltas per pixel (-1, 0, +1 in each of R, G, B). The whole image converges from 8-colour blocky toward full 16.7M-colour RGB.

Three properties that make this potentially better than monolithic diffusion:

1. **Tiny per-stage search spaces.** Each refinement step picks among ~27 deltas per pixel (vs monolithic RGB diffusion's continuous 24-bit-per-pixel search). Astronomically smaller per-step decision.
2. **No structural drift between stages.** The ±1-channel-level bound means the model literally cannot decide a wall is sky in stage 5 — structure locks in stage 1, later stages can only refine. This eliminates the "model loses its mind on long sampling chains" failure mode.
3. **Model size can shrink per stage.** Stage 1 has to solve composition + lighting + structure — needs a medium model. Stages 2-7 are local refinement — tiny models suffice.

Estimated compute saving: each step is much cheaper than a monolithic-RGB step due to the smaller search space. Total step count: 7 (one per rung increase) vs **35 monolithic DDPM steps** or **4 DMD-distilled steps**. End-to-end potentially **2-5× faster than a 35-step monolithic** at similar quality, with the bonus of bounded structural drift.

Adjacent literature:

- **Bit-plane coding** (JPEG2000 family) does progressive bit-depth refinement for *compression*, not generation
- **Cascaded Diffusion** (Imagen) cascades by *resolution*, not bit depth
- **SDEdit / bounded-noise diffusion** does a *single* bounded refinement, not a ladder of progressively-tightening ones
- **Wavelet diffusion** decomposes by frequency

To our knowledge, **the specific combination of (a) explicit progressive bit-depth ladder + (b) shrinking ± constraints between stages + (c) applied to generative diffusion is not in published literature** as of mid-2026. Pointers welcome if we missed something.

Open question: can a single trained model handle multiple stages of the ladder with timestep conditioning, or are separate models per stage required? The bounded-deviation property suggests separate small models per stage would specialise well and stay small.

C. How A and B compose with each other and with the UVW work

The two future directions combine cleanly with the world-coord canonical encoding from §4.1 to form a two-axis decomposition that may be the architecture of a fully UVW-native sibling model:

Axis	What it solves	Representation	Diffusion stages
Position (§4.1)	Where pixels live in 3D	UVW canonical channel (24-32 bit/pixel)	~4 large-noise steps
Per-channel bit-depth ladder (B)	What colour each pixel is	RGB at progressively-higher bit depth, 1 BPP → 8 BPP per channel	7 small-noise refinement steps

Each axis is structurally smaller than the combined RGB problem. Each can be trained / distilled independently. The total compute may be substantially less than one monolithic diffusion model that does everything at once.

For Lyra 2 specifically: the existing pipeline (`Sparse3DCache` → `_build_canonical_spatial_coords` → DiT → VAE-decode → 3DGS lifter) could be reorganised around this decomposition by training a sibling model that's structurally UVW-then-bit-depth-ladder rather than the current monolithic "predict RGB from coord + reference latent." The bit-depth ladder operates **per channel uniformly** — there's no luma/chroma split because RGB is intrinsically three independent intensity channels, not a luminance signal with chrominance side-channels.

D. Entity-structured prompting with a variable-byte semantic vocabulary

Orthogonal to A, B, and C: the caption that conditions Lyra 2's T5 encoder is freeform text today. Replacing it with a **structured entity-tag stream + expanded canonical descriptions** gives the model narrower priors at inference time and lets the same vocabulary act as a per-voxel semantic ID for downstream rendering.

The encoding (LEB128-style):

```
byte 0 msb = 0:               1 byte total, IDs 0..127         (top 128 frequent)
byte 0 msb = 1, byte 1 msb = 0: 2 bytes total, IDs 128..16511  (next 16k)
byte 0,1 msb = 1, byte 2 msb = 0: 3 bytes total, IDs 16512..2M (rare)
```

The data class:

```
@dataclass
class SemanticEntity:
    name:        str           # canonical id, e.g. "dolphin_bottlenose"
    string:      str           # prompt expansion text
    color:       (int,int,int) # baseline RGB hint
    scene_count: int           # cumulative usage across all generated scenes

    def __lt__(self, other):
        return (-self.scene_count, self.name) < (-other.scene_count, other.name)
```

`scene_count` drives **self-tuning byte-width assignment**: the vocabulary resorts periodically, and the 128 most-used entities get the 1-byte fast path automatically. Sci-fi-heavy projects end up with `nebula` / `station` / `asteroid` at 1 byte; fantasy with `tree` / `castle` / `dragon` instead. No hand-curated priority list.

Average storage per tag for a typical project: ~1.3 bytes.

Four uses, one vocabulary:

Pipeline stage	Encoding used
Caption parsing input (text → entity tokens)	variable 1-3 byte
Lyra 2 Lite conditioning embedding (entity → vector)	learned embedding at 2-byte cap (50 M params, LoRA-budget)
Per-voxel semantic atlas (output)	variable 1-3 byte, stored in voxel alpha channel
Downstream segment renderer (G-Buffer ID)	16-bit = 2 byte

The `color` field also feeds the diffusion as an **auxiliary spatial colour prior** — voxels tagged with `palm` get a green-brown baseline before the model paints them, nudging diffusion toward correct colours faster.

Expected quality / speed effect (based on conditional-generation literature):

- ~10% quality improvement at fixed compute (structured tokens reduce ambiguity in the caption space, less drift across the autoregressive cache)
- ~1.2× faster convergence per chunk (narrower search space)
- Compute-free spatial colour prior boosts cross-scene cache hit rates (the 1-byte fast-path entities are by construction the most-reused)

Implementation status: the `EntityVocabulary` class + the `StructuredPromptBuilder` wrapper + `lyra2_caption_hook` integration point + 74-entity seed library ship in the DownToEarth repo as of 2026-05-20. Drop-in replacement for raw captions; two-line integration into any Lyra-2-shaped inference script.

E. Verified accelerator stack (DMD + TeaCache + torch.compile + fp8 + SAGE)

The proposal sections above are forward-looking architectural ideas that require training compute we don't have. **This section catalogues the inference-time accelerator stack that is shippable today** with no retraining, drawing on mid-2026 SOTA we surveyed independently of this proposal.

Technique	Speedup	Status	Source
Lyra 2's bundled DMD distillation (35→4 steps)	**13×**	✅ already in `checkpoints/lora/dmd_distillation.safetensors`	nvidia/Lyra-2.0 HF
TeaCache (content-aware step skipping for Wan-family DiTs)	**2-3×**	✅ official Wan 2.1 support since Mar 2025, -0.07% VBench	ali-vilab/TeaCache
torch.compile + CUDA graphs on sm_120	**1.3-1.8×**	✅ requires flash-attn ≥ 2.8.3 to graph-fuse cleanly	PyTorch sm_120 PR #159207
fp8_e4m3fn storage + **SageAttention** on Blackwell	**1.8×**	✅ implemented in PR #61's `low_vram.py` (storage); TE compute-fp8 is ~1 week extra eng	Spheron Blackwell benchmark
Entity-structured prompting (D, above)	quality + 1.2×	✅ implemented	this proposal

Multiplicative product: `13 × 2.5 × 1.5 × 1.8 = ~88×` raw. Discounting for non-orthogonality between DMD and TeaCache (both save diffusion steps; some overlap) gives a **realistic ~60× speedup over stock Lyra 2** with no retraining.

What this does to the wall-clock budget on consumer hardware:

Hardware	Stock Lyra 2 + DMD (today)	60× stack
H100 SXM (cloud, ~$3.50/hr)	36 acres / 12 hr bake	**143 acres / 12 hr**
RTX 5060 Ti (consumer, $400)	3.5 acres / 12 hr bake	**17 acres / 12 hr**
Cost to bake 1 km² (Mêlée-class) on H100	~$1,400	**~$73**

Excluded from this stack (mutually exclusive with Lyra's own DMD, or research-direction only):

- CausVid / Self-Forcing / FastWan — alternative distillations, pick one
- LCM consistency distillation — superseded by DMD2 for DiT video
- 14B → 8B parameter distillation — months of training, not publicly available
- PyramidalWan multi-resolution finetune — promising (~2-3× on top) but needs ~$1k cloud compute and 1 month of fine-tuning

What we're NOT claiming

These are forward-looking notes, not measured results. Specifically we are NOT claiming:

- That any of these architectures would match Lyra 2's documented quality
- That the compute savings estimates above are accurate (they're back-of-envelope)
- That the bounded-deviation refinement actually trains stably in practice (it's an open empirical question)
- That this is a better path than Lyra 2's existing approach for the goal Lyra 2 was designed for

What we ARE claiming: these are coherent architectural ideas, novel in their specific combination, that emerge naturally from the UVW-bijection work and that we think are worth recording publicly in case they're useful to the team or the broader community. We claim the ideas, not the validation.

Provenance

Both ideas surfaced in design conversations between MiLO and Opie during the development of PR #61, 2026-05-19 and 2026-05-20. We're recording them here so the timestamp is on the public record. If either turns out to be a productive research direction — by NVIDIA or otherwise — we'd appreciate a citation; if either is already a published paper we've missed, we'd appreciate the pointer so we can credit appropriately.

What this doesn't claim

- **Quality wins.** I don't have the compute to retrain Lyra 2 and measure SSIM / LPIPS / FID. The argument is architectural: fewer things for the cross-attention to learn, smaller memory footprint for revisiting scenes. Whether that translates to measurable quality improvement is an open empirical question, and entirely depends on retraining infrastructure NVIDIA has and others don't.
- **A drop-in PR.** A retrain is required. This is a proposal to *consider* the encoding shift in a future training run, not a patch.
- **Replacing Sparse3DCache.** The frame-keyed cache has legitimate uses (temporal weighting, multi-view RGB storage). The world-coord atlas could complement it rather than replace it.

Open questions for the team

1. Has the team explored world-coord-keyed conditioning during Lyra 2's design? If yes, what failed?
2. The current `_world_points` storage is already world-coord per pixel — at what cost would the cross-attention be restructured to query that directly?
3. For the long-horizon unbounded case, is a hierarchical sparse atlas competitive with the current frame-keyed approach in terms of memory growth as the camera trajectory length grows?
4. The 8-bit quantization is the simplest version; 16-bit and 32-bit variants are documented in DownToEarth/README.md → Scaling section. Which precision tier would matter for Lyra 2's training data distribution?

About the proposers

This grew out of building a Lyra-2-inspired local pipeline (voxgaussian) that runs on consumer GPUs (single iter, ≈ 5 min from a single Juggernaut XL render to a 167 k-Gaussian splat cloud). We can't match Lyra 2's quality — the gap is the 14 B-param Wan-2.1 backbone trained on 64 × GB200s, not the architecture — but the UVW↔RGB bijection landed as a clean little data structure with one property that struck us as worth surfacing upstream: **the bytes are the coords**, both ways, at zero VRAM, and that property is rare enough to be useful beyond our tiny project.

Happy to discuss, refine, or retract any of the above. If the encoding shift turns out to be a bad fit, the comparison itself is still useful data for anyone considering similar architectures.