← SCENE DEMO | UVW EXPLAINER | SOURCE ON GITHUB

Down To Earth

🤝 **Upstream contribution — `nv-tlabs/lyra#61` — *draft RFC***

Architectural proposal filed against NVIDIA Toronto's Lyra 2 repo: shift the canonical-coord conditioning from frame-keyed `(u, v, frame_slot)` to a bidirectional world-coord ↔ RGB atlas. 312 lines of opt-in patches, DCO-signed, byte-identity-preserving in the default path. Includes a free wall-clock win for `Sparse3DCache.retrieve()` (the `octant_prefilter` kwarg) that requires no retraining and could land on its own.

Full writeup: LYRA2_PROPOSAL.md · PDF: LYRA2_PROPOSAL.pdf

🎮 **Live demos:**

- 🖼️ **Photoreal scene** (flagship): https://downtoearth-9lq.pages.dev *(167,192 colored Gaussians, baked from a single Juggernaut XL render through Hunyuan3D → voxel refinement → feed-forward splat fit. Toggle PHOTOREAL ↔ UVW to see the same data through the bijection.)*
- 📐 **UVW bijection explainer**: https://downtoearth-9lq.pages.dev/uvw *(Procedural hamlet scene, side-by-side panes showing the summary-atlas-class-palette rendering next to the canonical-RGB rendering. Both panes are the same data, just different shaders.)*
- 🧊 **UVW voxel raymarcher** (new 2026-05-20): `voxgaussian/voxel_renderer/index.html` *(Single-file WebGL2 demo of the runtime half of the Lyra-2-Lite pipeline. DDA voxel traversal + per-chunk frustum culling running on a 256^3 procedural island/tower/caverns scene at 60+ fps on a 5060 Ti. Press F to toggle culling, G to see culled chunks. Open locally — no server needed.)*

📄 **Read offline / share:**

- README.pdf (~1 MB · 9 pages)
- LYRA2_PROPOSAL.pdf (482 KB · the architectural RFC)
- 👴 **DUMMIES.pdf** (510 KB · *full project explained without computer-talk, for non-technical readers*)
- Markdown-rendered on .dev: /readme · /proposal · /dummies

A JRPG-style WebXR walker plus the **bidirectional voxel ↔ RGB atlas** data structure we built for it. The walker is the playground; the atlas is the headline. Final Fantasy VII / IX-style fixed-camera scenes — but the scenes are actual 3D geometry generated locally from AI-drawn stills, so you can walk freely on Quest 3 (VR or AR passthrough) or any modern desktop browser.

What is this?

← SCENE DEMO UVW EXPLAINER SOURCE ON GITHUB

🧑 In plain English	🤓 Technical
We give an AI a sentence ("a cobblestone village square at dusk") and it draws a single picture. Then we use *more* AI to figure out the 3D shape behind that picture — which pixels are ground, which are walls, how far away each thing is. From that we build a tiny voxel world (think Minecraft blocks, but very small ones) that you can actually walk around inside on a VR headset. To keep track of millions of those tiny blocks fast, we invented a special "index trick": every block gets a 3-digit address, and every color on the screen is also a 3-digit number — so we just say *the color IS the address*. You see a red-greenish-blue pixel? Those three numbers tell you exactly which block in the world you're looking at. No translation needed, no maths.	Pipeline lifts SDXL-generated stills (Juggernaut XL v9) to walkable 3D scenes via Hunyuan3D-2 multi-view mesh synthesis + Depth-Anything V2 walkable-polygon extraction. The `voxgaussian/` sub-pipeline refines the result iteratively: diffusion-guided voxel-occupancy with **per-voxel class-id histograms**, active-view-selection by uncertainty, ControlNet depth+semantic inpaint feedback, and feed-forward Gaussian-splat fitting on the converged grid. The data-structure backbone is a **bidirectional voxel ↔ RGB atlas** with mathematically-inherited identity: `(u, v, w) ⇔ (atlas_x, atlas_y)` is pure arithmetic, and `(R, G, B) ⇔ (u, v, w)` is byte-identity (the bytes in a rendered RGB framebuffer literally *are* the voxel coords). Renders on commodity GPUs, runs on Quest 3 Meta Browser, MIT licensed.

The UVW atlas (the headline)

🧑 In plain English	🤓 Technical

Imagine you have a giant Excel sheet, 4096 rows by 4096 columns. That's 16.7 million cells — which happens to be exactly the number of blocks in a 256×256×256 world (because 256 cubed equals 4096 squared, a tidy mathematical coincidence we exploit).
 	
We assign each block a coordinate triple like `(73, 12, 200)` — its position in the world. Then we lay them all out on the Excel sheet in a specific pattern: the first 16 rows hold blocks where the third number is 0–15, the next 16 rows hold
	A 256^3 voxel grid fits perfectly into a 4096^2 atlas because $256^3 = 4096^2 = 16{,}777{,}216$. We use a **16×16 tile layout**: each tile is one full 256×256 w-slice, arranged 16-wide across the atlas. Forward mapping:
	
	`def voxel_to_atlas(u, v, w):`
	`  return ((w & 15) << 8) | u, ((w >> 4) << 8) | v`
	
	Inverse:
	
	`def atlas_to_voxel(x, y):`
	`  return x & 255, y & 255, (y >> 8) * 16 + (x >> 8)`
	
	Both are O(1), purely arithmetic, branch-free. Bijection verified exhaustively across all 16.7M pairs.
	
	Identity inheritance. Because the mapping is closed-form, the "identity atlas" — the texture whose pixel `(x, y)` holds the RGB `(u, v, w)` of the voxel it represents — never needs to be materialised. A 5-line GLSL fragment shader computes it per-pixel:

where the third number is 16–31, and so on.

Now the magic: we *also* write each block's coordinate INTO its own cell as a color — red = first number, green = second, blue = third. A block at `(73, 12, 200)` gets the color `(73, 12, 200)`. **The color and the address are the same three numbers.**

Why does this matter? Because rendering 3D on a screen produces colored pixels. If every surface in the world is painted with its own coordinate, then any pixel you can see on screen already tells you what block it belongs to. No raycasting. No "trace a line from the camera through the pixel and see what it hits." The pixel just says it.

And here's the kicker: we don't even need to *store* the Excel sheet. The pattern is so regular that a 5-line shader can compute "what coordinate does Excel cell `(x, y)` represent" or "what cell does coordinate `(u, v, w)` live in" on demand. Zero memory. The identity is *free*.

What we DO store — separately — is the useful stuff *about* each block: what kind of thing is it, how sure are we, how many times have we looked at it. Four bytes of "summary" per block. Total cost: 16 megabytes. Compare to the alternative (a full per-block class histogram stored sparsely) and we get the same

```
in vec3 vUvw;
out vec4 fragColor;
void main() {
    fragColor = vec4(vUvw / 255.0, 1.0);  // byte-perfect identity
}
```

Decode is byte-unpacking. Reading the framebuffer at any pixel yields the voxel coord directly — no inverse projection, no matrix inverse, no sparse-tree traversal:

```
const [r, g, b, a] = readPixel(x, y);
const [u, v, w] = [r, g, b];          // ← that's it
```

Lineage. This is the bidirectional, persistent form of three older graphics primitives:

- *G-buffer position pass* (deferred shading, Crassin et al. 2008)
- *Space-filling-curve volume packing* (GigaVoxels et al.)
- *Color-as-ID picking buffers* (OpenGL 1.x era)

What we contribute is the **bijective bidirectional pair** with mathematically-inherited identity, allowing O(1) lookup in either direction with zero atlas storage. Closely related to Lyra 2.0's "warped canonical coordinate" trick (NVIDIA, April 2026) but more general — Lyra renders these coords transiently per-frame from a 3D point cloud cache; we maintain a static, GPU-cache-friendly bijection that any consumer (shader, JS, Python) can use.

answers in one texture lookup instead of 256.

← SCENE DEMO UVW EXPLAINER SOURCE ON GITHUB

The four payload bytes

🥱 In plain English

For each block in the world we keep four numbers next to its coordinate. They answer the four questions you actually have about any block:

1. **What is it?** — A number from 0 to 10. 0 means "empty/sky," 2 means "grass," 5 means "wall," 7 means "tree," and so on. Just an ID.
2. **How sure are we?** — Some blocks have been clearly identified (the AI looked at them from many angles and all the votes agree). Others are still up for debate. This byte stores the percentage of votes the winning answer got.
3. **How well-observed is it?** — A block we've looked at 300 times is different from a block we've looked at 3 times, even if both are 100% sure. We store this on a log scale so a few extra observations for an undersampled block matter more than for an oversampled one.
4. **How controversial is it?** — When the top guess is "tree" with 30% and the runner-up is "wall" with 28%, that's a problematic block — we should look at it again. This byte stores the gap between the top guess and the runner-up.

These four together cover almost every question the rest of the system asks about a block, without ever needing to peek at the full vote history. They're the cheat sheet.

🤯 Technical

Summary atlas at each voxel stores **RGBA8 = (cls, conf, obs, mrg)**:

Byte	Meaning	Encoding
`R`	mode class id	`argmax(histogram)` ∈ [0, 10]
`G`	mode confidence	`top_count / total × 255`
`B`	observation count, log-scaled	`clamp(log2(total + 1) × 16, 0, 255)`
`A`	ambiguity margin	`(top_count − runner_up_count) / total × 255`

Why each byte earns its slot.

- `R` collapses class-palette lookup to a single sample + LUT — every renderer's most-asked question.
- `G` drives convergence detection (`refine.py` tolerance threshold) and revision gating ("below-confidence voxels stay open").
- `B` is the easy-to-miss one — without it, high-confidence-3-votes looks identical to high-confidence-300-votes, and active-view-selection over-trusts undersampled cells.
- `A` is the killer byte for active view selection. Priority becomes a single-pass reduce:

  ```
  priority = (1 − margin/255) × (1 − obs/255)
  ```

  High margin = confident decision (skip). High obs = well-sampled (skip). Both low → controversial AND undersampled → point a camera there next.

← SCENE DEMO UVW EXPLAINER SOURCE ON GITHUB

Consumer mapping — every downstream subsystem becomes a single texture sample:

Consumer	Reads	Cost
Live viewer	R + G	1 sample → palette → display
Ray-carver	R + G + B	1 sample, 3 booleans
Active view select	A + B	1 reduce-sum over frustum
Convergence detector	G or A	histogram of the summary
Phase B (texture pass)	R	class-conditional texture choice

Full per-class vote histogram (the source of truth) lives separately in a `Texture2DArray<R8>[4096 × 4096 × n_classes]` — read-cold, written only during refinement passes. Summary atlas is the read-hot fast path.

How the demo works

👶 In plain English	🤯 Technical
Open the live demo and you'll see two views of the same little village side by side. The **left view** is the village painted "normally" — green for grass, brown for buildings, dark green for trees. Looks like a tiny voxel diorama. The **right view** is the village painted with each block's address as its color. So the ground at the front-left looks dark red (low first number), the ground at the back-right looks bright cyan (high second and third numbers). It looks weirdly psychedelic — but every speckle of color is a *meaningful* coordinate. Now move your mouse anywhere over the scene. The info panel in the top-right updates in real-time. It shows: • Where your cursor is on the screen	Single Three.js scene, two `setViewport`-separated panes, shared `InstancedMesh` of ~20,000 cubes. Each instance carries a `vec3 instanceUvw` attribute holding its voxel coord. **Left pane (summary):** `ShaderMaterial` whose fragment shader runs `voxelToAtlasUV(vUvw)` (5 lines of GLSL), samples the `summaryAtlas` DataTexture, casts `R × 255` to class id, indexes a `vec3 palette[11]` uniform, modulates brightness by `G` (confidence) for visual variety. **Right pane (canonical):** `ShaderMaterial` whose fragment shader is literally: `gl_FragColor = vec4(vUvw / 255.0, 1.0);` **Hover decode pipeline:**

- The exact RGB color under your cursor on the right view
- The voxel coordinates of that block in the world

And here's the proof — those last two rows are always **identical numbers**. The byte values you can see on screen ARE the voxel coordinates. There's no math hiding anywhere.

Below that, the panel tells you what class the block is (with a colored swatch), how confident we are (as a percentage), how many observations we've collected (on a log scale, like "≈2^4.0"), and how controversial the block is. All looked up from the summary cheat sheet in one fast texture read.

← SCENE DEMO UVW EXPLAINER SOURCE ON GITHUB

1. Animation loop renders the canonical pass to an offscreen `WebGLRenderTarget` (RGBA8, nearest-filter) each frame, with `scene.background = null` and grid hidden so non-voxel pixels stay `(0, 0, 0, 0)` and the alpha cleanly distinguishes hit vs miss.
2. `mousemove` handler computes WebGL pixel coords (origin flip) and calls `renderer.readRenderTargetPixels(rt, x, y, 1, 1, buf)` — one pixel readback, synchronous.
3. The bytes `(r, g, b)` are the voxel coord. No transformation.
4. JS `Map<key=u|v<<8|w<<16, summary>>` lookup yields the summary entry.
5. HUD updates RGB, voxel, atlas pixel, class+swatch, and three filled bars for confidence / obs / margin.

Performance note: `readRenderTargetPixels` is a synchronous GPU stall. Fine for inspect-mode UI; for continuous tracking, switch to async readback via `PIXEL_PACK_BUFFER`. Not done — would be ~20 lines.

Procedural scene generation at page load: a tiny `buildSceneAndAtlas()` function bakes ~20k voxels (ground slab, diagonal path, four hollow buildings, eight trees, a well) into a real summary atlas using the same arithmetic as the Python pipeline. So the demo's atlas would be readable by `pipeline/uvw_atlas.py::build_summary_atlas()` and vice-versa.

voxgaussian — the broader pipeline

🧑 In plain English	🤓 Technical

The atlas is one piece of a bigger recipe. Here's the whole flow:

1. **Pick a place to make.** You write a prompt like "a misty forest clearing at dawn with a stone well."

```
┌──────────────────────────────────────────────────────────┐
│ Phase 0  ·  Bootstrap                                    │
│   prompt → Juggernaut XL v9 → 1024² scene PNG            │
│   PNG → Hunyuan3D-2 → triangle mesh                      │
│   PNG → OneFormer (ADE20K) → semantic segmentation       │
│   PNG → Depth-Anything V2 → per-pixel depth              │
│   → seed VoxelStore(256³) with per-voxel class histograms│
└──────────────────────────────────────────────────────────┘
                           │
                           ▼
┌──────────────────────────────────────────────────────────┐
│ Phase A  ·  Iterative voxel-occupancy refinement (× ~12 iters) │
```

2. **AI draws it.** Stable Diffusion (specifically a model called Juggernaut XL) produces a single high-quality photo of that place.
3. **AI guesses the 3D.** Another model called Hunyuan3D looks at the picture and produces a rough 3D mesh — the basic shape of the buildings, ground, trees.
4. **AI guesses the depth.** Yet another model called Depth-Anything V2 estimates how far away each pixel is from the camera.
5. **We voxelise.** Convert the rough 3D mesh + depth map into our block-world (the 256^3 grid). Each block gets initial guesses about what it is.
6. **We iteratively refine.** Pick a random angle to look at the world from, render what we currently think the world looks like, ask the AI to fix any uncertain bits, vote those answers back into the blocks. Repeat 12 times. Things that disagree work themselves out by majority vote.


```
        sample candidate cameras
        score by Σ (1 − margin) × (1 − obs_log) in frustum
        ← single pass over summary_atlas (`uvw_atlas.py`)

   2. render_voxels():
        rasterise voxels with two outputs
        − depth (front surface)
        − semantic (mode class via summary R-byte)

   3. ControlNet inpaint (SDXL + depth + semantic ControlNets):
        condition on rendered passes via uvw_demo.html's
        canonical-coord trick — feeds clean (u, v, w) channels
        instead of warped-RGB, sidesteps disocclusion artifacts

   4. propagate():
        unproject inpainted pixels back to voxels along the
        camera ray, vote class+confidence into histograms
        ray-carve empty space along ray up to first hit

   5. converge check: stop when
        Σ Δhistogram / Σ histogram < 0.02
```

```
Phase B  ·  Texture + Gaussian-splat fitting
  project original PNG colors onto converged voxels
  fit one or more Gaussians per occupied voxel
  export .ply for splat renderer, .glb for mesh renderer
```

```
Phase C  ·  WebXR delivery
  Three.js + InstancedMesh (OCCUPANCY mode)
  Three.js + ShaderMaterial billboards (GAUSSIAN mode)
  Three.js + canonical-pass + summary atlas (UVW mode) ← new
```

Resolution adaptivity. The UVW bijection is hard-coded to 256^3 in `pipeline/uvw_atlas.py`, but the viewer JS recomputes tile layout from `snapshot.resolution` (default 128^3, with multi-resolution coarse/fine in `MultiResolutionVoxels`). Any res ≤ 256 packs cleanly into ≤ 4096^2 with 1-byte-per-axis identity.

Snapshot format (transported over WebSocket from `pipeline/live_server.py` to the viewer):

```
row = [ix, iy, iz, cls, conf, r, g, b, ox, oy, oz, nx, ny, nz, obs, mrg]
```

The last two bytes (`obs` , `mrg`) were added 2026-05-19 as part of the UVW integration — backwards-compatible with viewers that only read 14 fields.

Things that nobody can see get carved away as empty space.

7. **We add texture.** Once the *shapes* settle down, project the original picture's colors onto the blocks so they look photographic rather than flat-colored.
8. **You walk around.** Open the viewer on your Quest 3 or laptop and stroll through the place.

The atlas trick is what makes steps 5, 6, and 8 fast.

← SCENE DEMO | UVW EXPLAINER | SOURCE ON GITHUB

Scaling — what extra bytes per channel buy you

The whole scheme hinges on a single property: **the RGB values rendered by the canonical pass *are* the voxel coordinates.** Doubling the bits per channel doubles the world size you can name without losing this byte-perfect identity.

Current tuning (what's deployed)

Setting	Value	Why
Atlas format	**RGBA8**	One byte per axis = 256^3 voxels = 16.7M cells, fits a single 4096^2 texture (64 MB VRAM). Byte-perfect identity: `readPixel(x,y)[0..3]` *is* `(u, v, w, payload)`.
Antialias	`MSAA on`	Cheap at 23k instances. Smooths cube edges. Since hover-decode is gone, MSAA at edges doesn't compromise anything.
Output color space	`LinearSRGBColorSpace` (demo)	Skips Three.js's linear→sRGB conversion so the canonical-pass bytes display *exactly* as the voxel coords. The main viewer keeps `SRGBColorSpace` because its OCCUPANCY/GAUSSIAN modes benefit from gamma-correct color.

Setting	Value	Why
Tone mapping	`NoToneMapping` (demo)	A filmic curve would shift the canonical-pass bytes. No curve = byte-faithful display.
`frustumCulled`	`false` on InstancedMesh	Three.js can't bound an InstancedMesh by its instance positions (only the base geometry). Computing the bound ourselves would cost more than the cull saves at our voxel counts.
Pixel ratio cap	`min(devicePixelRatio, 2)`	Higher only helps 4K+/Retina; Quest 3 already supersamples in compositor.

Estimated VRAM for the current setup: **~80–100 MB**. Estimated FPS: **vsync-locked** on any GPU made after 2018, **90 Hz solid** on Quest 3 in immersive VR. Both have plenty of headroom for the upgrades below.

The byte-depth ladder

👱 In plain English	🤯 Technical
We've got 4 bytes (R, G, B, A) per voxel right now — each in the range 0 to 255. That gives us **256 × 256 × 256 ≈ 17 million voxels**, which sounds like a lot but is actually only a 2.5-meter cube if each voxel is 1 centimeter. To make a *bigger* world we don't change the scheme — we just give each colour channel	We've got 32 bits of address space across 4 channels at RGBA8. The scheme generalises trivially to wider channels because the bijection itself is just `voxel_to_atlas(u, v, w) → (atlas_x, atlas_y)` arithmetic — it doesn't care about per-channel width.

Format	Per pixel	Per axis	Cube size @ 1 cm	Total addressable	Identity	GPU support
RGBA8	4 B	2^8 = 256	2.5 m	16.7M (2^{24})	byte-perfect	universal
RGBA16UI	8 B	2^{16} = 65,536	655 m	281T (2^{48})	uint16-perfect	WebGL2 + `EXT_color_buffer_integer`
RGBA16F	8 B	≤1024 (mantissa)	10 m	≈10^9 exact	lossy >1024	WebGL2 native
RGBA32F	16 B	2^{24} = 16.7M (exact)	167 km	4.7×10^{21}	exact ≤2^{24}, lossy past	WebGL2 native
RGBA32UI	16 B	2^{32} = 4.29B	42,000 km	7.9×10^{28}	uint32-perfect	WebGL2 + `EXT_color_buffer_integer`

Upgrade triggers:

- **256^3 → 1024^3 (RGBA16):** any scene larger than ~2.5 m at 1 cm voxels, or ~256 m at 1 m voxels. Realistic walkable scenes (hamlets, courtyards). Per-pixel atlas cost doubles to 8 B; 4096^2 atlas → 128

more bits. The address is still the colour; the colour is still the address. Just wider numbers.

16-bit per channel (RGBA16) doubles each axis up to 65,536 values. That's a 655-metre cube at 1 cm per voxel — enough for a village, a forest path, a small town. About **281 trillion** voxels addressable. The colour you see on screen is still literally the coordinate, just packed in two bytes per channel instead of one.

32-bit floats per channel (RGBA32F) take it to 16.7 million per axis — a **167-kilometre cube**, big enough for a

MB.

- **1024³ → 65k³ (RGBA16):** same format, just bigger atlas (sharded across a texture array since 65k × 65k = 4.29B pixels exceeds single-texture limits). VRAM stays bounded by what you actually populate.
- **65k³ → millions³ (RGBA32F or RGBA32UI):** city / regional scale. Storage becomes the constraint, not addressing. Need sparse virtual-tile data structures (à la GigaVoxels / Crassin) — coord space stays bijective within each leaf tile.
- **Beyond:** sparse hashed page table. The byte-perfect coord-as-RGB property is preserved *inside* each leaf tile; cross-tile lookups need an indirection.

Pick RGBA16UI over RGBA16F for any production use. The byte-perfect identity is what makes the whole scheme tractable for downstream consumers (ControlNet conditioning, picking buffers, picking-via-shader); fp16 mantissa precision compromises that above 1024.

Pick RGBA32UI over RGBA32F for the same reason, with the caveat that `EXT_color_buffer_integer` is desktop-only (Quest 3's Adreno 740 supports it; older mobile may not). If portability matters, RGBA32F with `floor()` decode is the safer pick.

For Gaussian splats specifically (one Gaussian per voxel-slot), RGBA16 addresses **~281 trillion potential splats** — orders of magnitude beyond what any production GPU can render (typical 3DGS scenes hit 1–100M; Lyra 2.0 trains on 90-metre walkable worlds at ~1B). The address space is never the constraint; **per-splat storage budget always is**.

small city through an open countryside. Total addressable: about **5 sextillion** voxels. The colour-is-coord property gets slightly fuzzy here (floats aren't bit-exact integers above 2^24) but for practical purposes still works.

32-bit integers per channel (RGBA32UI) take it to 4.29 *billion* per axis — that's a **42,000-kilometre cube** at 1 cm. Solar-system scale. Byte-perfect identity intact.

Past that, you'd be modeling Earth-scale environments — at that point you stop adding

bits and start being clever about *which* voxels you actually store (a sparse data structure: only the parts of the world that exist get a slot, not every theoretical position).	← SCENE DEMO UVW EXPLAINER SOURCE ON GITHUB

Per-voxel occupancy bit (the early-skip optimisation)

Independently of the byte-width family above, a **same-resolution 1-bit-per-voxel companion texture** is a worthwhile add for any sparse scene. The bit answers one question — "*is this voxel populated?*" — and lets a shader skip the multi-byte main-atlas read for empty voxels entirely.

Grid resolution	Voxel count	RGBA8 atlas (4 B/voxel)	RGB-only (3 B/voxel)	**1-bit occupancy mask**	Mask vs 3-byte ratio
64^3	262 K	1.0 MB	786 KB	**33 KB**	**24× smaller**
128^3	2.1 M	8.4 MB	6.3 MB	**262 KB**	24×
256^3	**16.7 M**	**67 MB**	**50 MB**	**2.1 MB**	**24×**
512^3	134 M	537 MB	403 MB	16.8 MB	24×
1024^3	1.07 B	4.3 GB	3.2 GB	134 MB	24×

The bitmap **compounds three wins**:

1. **Storage** — 24× less memory than 3-byte-per-voxel, 32× less than RGBA8
2. **Bandwidth** — 24× less data fetched per voxel during occupancy checks; on bandwidth-bound shaders (most raymarching loads), this roughly translates to 10-20× faster scans
3. **Cache locality** — 24× more voxels fit in L1/L2 GPU cache, so sequential scans hit cache vastly better

Where it really pays off: sparse scenes (~5% surface occupancy). Pairing a dense 1-bit mask with **sparse** RGB storage for populated voxels only:

```
Dense 1-bit mask + dense RGB:      52 MB (worst-case)
Dense 1-bit mask + sparse RGB (5% pop): 4.6 MB (typical voxgaussian scene)
All-sparse hash table (no mask):        4.0 MB (no O(1) addressing)
```

The mask version gets **O(1) addressing** (you can query any coord) plus **near-sparse-hash memory cost** (just the bytes you actually populated). Best of both for the access pattern of "raymarch through mostly-empty volume."

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;
}
```

Implementation shipped in `pipeline/uvw_atlas.py` as the `OccupancyBitmap` class — 8-voxels-per-byte X-axis packing for cache locality, doctested + verified inside the self-test (`python -m pipeline.uvw_atlas`). Loads directly from voxgaussian's sparse histogram dict via `OccupancyBitmap.from_voxel_store(...)` . Memory: 2 MB at 4096^2 atlas (256^3 voxels), 32× smaller than the RGBA8 summary atlas.

Extension: 2-bit-per-voxel for demand-driven streaming (2026-05-20)

The bitmap class now also supports a **2-bit mode** that carries occupancy *and* a per-frame "render-flag" bit (touched by a ray this frame). Pass `bits_per_voxel=2` at construction. Storage doubles (4 MB at 4096^2 atlas — still trivial), and the runtime gains a precise log of which voxels actually contributed to a pixel that frame.

The render-flag is bijection-keyed. The render-flag bit for voxel `(u, v, w)` lives at the **same atlas address** as that voxel's RGBA data — both addressed via the UVW↔RGB bijection. A single `voxel_to_atlas(u, v, w)` call gives the raymarcher one coordinate that serves both the colour read and the flag write. New voxel-coord helpers (`set_by_voxel` , `set_touched_by_voxel` , etc.) make this bijection use first-class in the API.

```
bmap = OccupancyBitmap(atlas_w=4096, atlas_h=4096, bits_per_voxel=2)
bmap.set(ax, ay)              # occupancy (baked offline)
bmap.set_touched(ax, ay)      # raymarcher: ray terminated here
chunks_needed = bmap.touched_chunks(chunk_size=16)
                              # {(cx,cy,cz)} -- drives streaming
bmap.clear_touched()          # masked AND, preserves occupancy
```

This is the data structure for demand-driven sparse voxel streaming — the same pattern UE5 Nanite uses for triangle clusters and id Tech 5 MmegaTexture uses for 2D textures, applied to voxels. Per-frame loop:

```
clear_touched()                 # ~10 µs
render_frame()                  # raymarcher imageAtomicOrs the touch bits
touched = touched_chunks()      # OR-reduce to chunk keys
streamer.update(touched)        # SSD fetch + eviction
```

Critically, this is omnidirectional / VR-correct because the touched bitmap is populated by *whatever rays were cast* — per-eye VR frustums, equirectangular 360° spheres, planar FPV — all populate correctly. No view-direction assumption baked in. Frustum culling would mispredict on VR head rotation; demand-driven streaming makes no such mistake.

The runtime renderer (when wired) calls `imageAtomicOr` on a `uimage2D` bound to the bitmap during the per-pixel raymarch. CPU-side, the `touched_chunks()` reduction takes ~0.5 ms for a 4 MB bitmap and yields the streaming-input set in one call.

Generation budgets at Lyra-2-Lite scale (2026-05-20)

The UVW atlas / occupancy bitmap make a chunked voxel world *renderable* on consumer hardware. The complementary question is how much *content* you can bake into that world per dollar of compute. The proposal's "Lyra 2 Lite" architecture (UVW canonical conditioning) on top of the verified mid-2026 accelerator stack (DMD distillation + TeaCache + torch.compile + fp8 + SAGE-attention) projects to **~60× speedup over stock Lyra 2** with no retraining:

Target territory	Real-world size	H100 hours (Lyra 2 Lite mixed)	Cost @ $3.50/hr
Apartment	100 m^2	0.02 hr	$0.07
Single building / shop	500 m^2	0.10 hr	$0.36
Adventure-game village	5,000 m^2 (~1 acre)	0.7 hr	**$2.45**
Mêlée-class island	1 km^2	21 hr	**$73**
Caribbean archipelago	5 km^2	104 hr	$364
Small district	50 km^2	1,040 hr	$3,640

On a single 5060 Ti at home (no cloud):

Target	Wall time
Village (1 acre)	~1.5 hr
Mêlée-class island (1 km^2)	~7.2 days continuous
5 km^2 archipelago	~36 days continuous

The full breakdown of the accelerator stack, sourced from mid-2026 SOTA (TeaCache by ali-vilab, DMD shipped in `nvidia/Lyra-2.0`, PyTorch sm_120 + flash-attn ≥ 2.8.3, NVIDIA Blackwell fp8 native), lives in LYRA2_PROPOSAL §6.6.E.

The runtime side (the voxel raymarcher + chunk streaming + 2-bit OccupancyBitmap demand-driven loading) renders these baked worlds at 60+ fps on the same 5060 Ti that did the bake. **Offline bake costs dollars; online play costs nothing.**

How a TB-scale world raymarches from disk

← SCENE DEMO UVW EXPLAINER SOURCE ON GITHUB

The UVW atlas + 2-bit OccupancyBitmap aren't just compact data formats — they're the foundation for **rendering arbitrarily-large voxel worlds on consumer hardware**. A 16 GB graphics card can render a 5 TB world at 60+ fps, not by holding it all in VRAM but by holding the **right 6 GB** at every moment. This section explains how.

The paradox

World scale	Total voxels (1 cm)	RGBA atlas size	VRAM on 5060 Ti
Bedroom (3 m^3 × 1 cm)	~3 million	12 MB	trivial
Apartment (100 m^2 × 2 m × 1 cm)	~2 billion	8 GB	tight
Small island (1 km^2 × 50 m × 1 cm)	**5×10^{10}**	**200 GB**	**doesn't fit**
Town (10 km^2 × 80 m × 1 cm)	8×10^{11}	3.2 TB	doesn't fit
City (100 km^2 × 100 m × 1 cm)	10^{13}	40 TB	doesn't fit
Microsoft Flight Sim's Earth	~10^{15}	**~2.5 PB**	obviously doesn't fit

Every entry past "apartment" exceeds VRAM by orders of magnitude. Yet *Flight Sim* runs at 60+ fps over the entire planet, *Teardown* simulates voxel destruction in city-block scenes, *UE5 Nanite* renders billion-triangle worlds. The trick is the same in all three: **only the bytes that contribute to a visible pixel need to be in VRAM at any moment.**

The fundamental insight: pixels, not voxels

At 1080p, the screen has 2,073,600 pixels. At 4K, 8.3 million. **No matter how big your world is, you'll never render more than that many voxels per frame** — one per pixel. The rendering cost scales with screen resolution, not with world size.

So the question becomes: how do you get the **right** ~2 million voxels into VRAM each frame? Not the world's worth — just the camera-visible subset.

The math:

Resolution	Pixels per frame	Voxels touched	At 32 bytes/voxel	Bytes/frame
1080p	2.1 M	~5 M (with raymarch steps)	32 B	**160 MB**
4K	8.3 M	~20 M	32 B	**640 MB**
8K	33 M	~80 M	32 B	2.5 GB

A 1080p frame's *visible voxel set* is 160 MB. That's the **working set the GPU actually needs.** Everything else — the other 199.84 GB of the "small island" world — sits on disk, untouched, waiting for the moment the camera moves into a position that needs it.

What lives where

Across the memory hierarchy, the budget breakdown for a Melée-class 1 km² world on a 5060 Ti:

Tier	Capacity	Holds	Latency
GPU registers / L1	~1 MB	current ray's voxel + neighbours	<1 ns
GPU L2 cache	~32 MB on Blackwell	last ~1M voxels accessed	~10 ns
VRAM (working set)	**6 GB / 16 GB total**	**~200 16^3-voxel chunks** = recently-touched scene area	~200 ns
System RAM (page cache)	32 GB	**~1 GB of warm chunks** waiting to upload to VRAM	~100 µs
NVMe SSD	2 TB	**the full 200 GB world atlas**	~50 µs random read
HDD or cloud	unlimited	cold tiles you may never reach	~10 ms

The hierarchy works because **each tier holds only what the tier below it just used recently** plus some predictive lookahead. VRAM holds chunks the player is currently in or about to walk into; RAM holds chunks they might walk into in the next few seconds; SSD holds everything.

Frame-by-frame anatomy

Here's what actually happens to render one 16.67 ms frame at 60 fps:

```
T = 0 ms        Frame starts
                bitmap.clear_touched()                    [~10 µs]
                  AND-mask preserves occupancy, zeros touched bits

T = 0.01 ms     GPU raymarcher launches 2.1M pixel rays
                Each ray DDAs through voxel space:
                  - Sample 2-bit OccupancyBitmap entry → cheap
                  - If empty: skip via DDA stepping
                  - If occupied AND chunk-in-VRAM: read RGBA, shade
                  - If occupied AND chunk-NOT-in-VRAM: read coarse-LOD
                    fallback voxel (always-resident pyramid), atomic-or
                    the "touched" bit so the streamer knows we wanted
                    this chunk
                  - Either way: set bit-1 in voxel state (touched)

T = ~10 ms      Render complete, present to display

T = ~12 ms      Background CPU thread:
                  touched = bitmap.touched_chunks()       [~0.5 ms]
                    → set of (cx, cy, cz) keys
                  diff against current_resident_chunks    [~0.1 ms]
                  for each new chunk not yet resident:
```

```
                    queue NVMe read (async)
                  for each chunk not touched for K=60 frames:
                    schedule eviction

T = ~15 ms      NVMe DMA: 1–3 new chunks land in VRAM
                  Typical chunk: 16³ × RGBA8 = 16 KB
                  3 chunks = 48 KB = 1 NVMe block

T = 16.67 ms    Next frame starts
```

Key observation: **the touched bit and the SSD streaming run in parallel with rendering, not in serial.** The renderer never blocks on the SSD. The renderer's responsibility is to *say what it needed* (via the touched bit); the streamer's responsibility is to *have it ready for next frame*. Cold-cache misses fall back to the LOD pyramid, which is always resident.

Bandwidth math: does it actually fit?

Per frame, with the player moving at typical FPS-game speed (5 m/s):

```
Camera advances:        5 m/s × 0.01667 s = 8.3 cm/frame
Voxels at 1 cm:         ~8 new voxels along view direction per frame
Chunks at 16³ voxels:   ~0.5 new chunks/frame (statistical)
At 60 fps:              ~30 chunk reads/sec, ~480 KB/sec
NVMe Gen4 read:         ~5 GB/sec
Headroom:               ~10,000×
```

So for typical movement on 1 cm voxels, **NVMe is wildly over-provisioned**. The bandwidth budget supports much higher movement speeds (running, vehicles, teleport) without any change.

For aggressive cases (teleport, fast head rotation in VR):

```
Teleport (instant pos change):  ~200 new chunks needed = 3.2 MB
NVMe read of 3.2 MB:             ~0.6 ms
At 60 fps frame budget:          16.67 ms
                                  → still fits in one frame
```

Even a worst-case teleport pulls < 1 ms of NVMe time on a Gen4 SSD. As long as the LOD pyramid covers the first frame's pixels with coarse-but-correct data, the high-detail chunks stream in within one frame and the user never sees pop-in.

LOD: scaling beyond bandwidth limits

At very large scales (10+ km² worlds) or very high movement speeds, the NVMe bandwidth eventually becomes the bottleneck. Solution: **a resolution pyramid**, mip-style:

LOD level	Voxel size	Storage for 1 km²	Used at distance
0 (full)	1 cm	200 GB	0-30 m (close-up walking)
1	2 cm	25 GB	30-60 m (mid-field)

LOD level	Voxel size	Storage for 1 km²	Used at distance
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 (vista)	32 cm	6 MB	600+ m (always resident)

Coarser levels are smaller and naturally always-resident. The ray-driven streamer picks the LOD level matching the **screen pixel size at that depth**, so detail you can't see is never fetched. This is the standard mip-mapping trick from textures, applied to voxels.

With LOD active, even a **10 TB world** has only ~50 GB worth of "high-detail close-up chunks" total, and at any moment the player is within 30 m of only ~6 GB of them. Working set stays bounded regardless of world size.

Worked examples at different scales

Scale	Total disk	LOD-pyramid disk	VRAM working set	NVMe stream rate (walking)
Apartment (8 GB)	8 GB	9 GB	6 GB	~50 KB/s
Mêlée island (1 km²)	200 GB	**230 GB**	**6 GB**	~500 KB/s
Caribbean chain (5 km²)	1 TB	1.15 TB	6 GB	~500 KB/s
Town (10 km²)	2 TB	2.3 TB	6 GB	~1 MB/s
City (100 km²)	20 TB	**23 TB**	**6 GB**	~2 MB/s
Continent (10,000 km²)	2 PB	2.3 PB	6 GB	~5 MB/s

The VRAM working set is constant across all rows. World size multiplies disk consumption linearly; it doesn't affect what's resident. The 5060 Ti's 16 GB caps the *detail-near-camera budget*, not the *world size*.

Hard limits and what breaks

1. **Random teleport to a brand-new region** — the first frame after the teleport has only the LOD pyramid available (no high-detail chunks loaded yet). For ~one frame the world looks slightly less crisp at close range. Fully resolves in <50 ms.
2. **Walking faster than NVMe can stream** — at >100 m/s (~360 km/h) sustained, the chunk fetch rate becomes the bottleneck even on Gen4 NVMe. Mitigations: bigger LOD-1 always-resident layer, reduce LOD-0 chunks in motion-aligned direction, or accept slight detail blur during fast travel.
3. **VR head rotation faster than 720°/sec** — beyond which the touched chunks change faster than NVMe can refresh. Realistic human head rotation peaks at ~500°/sec; well within budget.

4. **Worlds bigger than your SSD** — needs cloud-backed storage (Cloudflare R2 / S3 / similar) with a local SSD acting as a sliding-window cache of the wider world. *Flight Sim* does this: 2.5 PB lives in MS's data centres, your local SSD caches the chunks for the region you're currently flying over.
5. **Dynamic objects** — not addressed by this architecture. Moving characters, particles, doors need a separate layer (small per-object voxel atlases with transforms) composited against the static raymarch. See LYRA2_PROPOSAL §6.6 for the future-direction notes on this.

Prior art

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

System	Domain	Year	What gets paged
id Tech 5 MegaTexture (Rage)	2D textures	2011	Texture pages (whatever's rasterized)
id Tech 6 Virtual Texturing (Doom 2016)	2D textures	2016	Generalized MegaTexture
NVIDIA GVDB	Sparse voxels	2018	Voxel pages
Unreal Engine 5 Nanite	Triangle meshes	2022	Cluster pages
Microsoft Flight Simulator 2024	Earth-scale terrain	2024	Cloud-streamed regional tiles

The novelty in DownToEarth's application isn't the streaming pattern — it's the combination with:

- **UVW bijection** (the chunk's address-key IS its rendering color channel, no separate index lookup)
- **2-bit OccupancyBitmap** (occupancy + touched in one structure, driving both render-skip and streaming decisions)
- **Lyra-2-derived content** (the voxel data itself is offline-baked from a diffusion model, so the world generation pipeline feeds the same streaming format)

The result: a **content-creation + rendering pipeline that's internally consistent** from the AI bake all the way to the per-pixel ray. The streaming voxel renderer isn't bolted on to an unrelated generator; the generator's output format is the renderer's input format.

Per-channel payload split (the orthogonal axis)

The bits don't have to go entirely into the *address*. The split between address bits and payload bits is independent:

Strategy	Address bits	Payload bits	Use case
Pure address (RGB-coord, A-payload)	24	8	One class byte per voxel — current scheme

Strategy	Address bits	Payload bits	Use case
Spatial-only address, 4 payload bytes	24	32	Class + confidence + obs + margin all packed (the actual current summary atlas)
4D address (RGBA-coord)	32	0	Time, scene-id, or class-histogram-bin as 4th dim. Payload lives elsewhere.
Hybrid (RGB-coord, A as class) + sibling texture for confidence	24	8 + payload texture	Decoupled. Standard for production pipelines.

The current code uses the second strategy: 24 bits of address (one byte per axis), 32 bits of payload (mode/conf/obs/margin). At RGBA16 you'd have 48 bits of address and 64 bits of payload — enough to pack class+confidence+observation-count+ambiguity-margin at full 16-bit precision per byte, or to store half-float Gaussian-splat parameters directly in the atlas without a sibling texture.

Prior art and credit

This work recombines well-known graphics primitives in a way that's useful for diffusion-guided voxel-occupancy refinement. None of the underlying ideas are novel; the combination and application are.

Primitive	Origin	What we do differently
G-buffer position pass	Crassin et al. 2008; every deferred renderer since	Make it a static, persistent atlas instead of a per-frame transient; 8-bit per axis instead of float32
Space-filling-curve volume packing	GigaVoxels (Crassin 2009), sparse virtual textures	Use a simpler 16×16 tile layout (not Hilbert/Morton) since $256^3 = 4096^2$ exactly; preserve spatial locality at the slice level
Color-as-ID picking buffer	OpenGL 1.x era, every editor	Make it bidirectional and bijective rather than just forward (color → pick)
Per-frame canonical-coord ControlNet conditioning	Lyra 2.0, NVIDIA Toronto (April 2026)	Static atlas-based instead of inverse-warped from a 3D point-cloud cache; works in deterministic shader instead of via learned cross-attention
Active view selection by uncertainty	"Next-best-view" literature (Connolly 1985, many since)	Reduce to a single-pass scan over `(margin, obs)` bytes of the summary atlas
Per-voxel class histograms for outlier rejection	Common in vision (e.g. RGB-D fusion)	Persist as a `Texture2DArray<R8>` slice-per-class so vote increments are GPU-atomic

Not derived from NVIDIA Lyra 2.0. Lyra 2.0 ships under a research-only license. This repository is original work, MIT licensed. The conceptual overlap is acknowledged above, no code or weights from Lyra are used.

Useful references:

- GEN3C (NVIDIA, CVPR 2025) — arxiv:2503.03751 — the "3D cache rendered to condition video gen" pattern
- VMem (Oxford, ICCV 2025) — arxiv:2506.18903 — surfel-indexed view memory for autoregressive scene gen
- Lyra 1.0 (NVIDIA, Sept 2025) — arxiv:2509.19296 — video diffusion + feed-forward 3DGS lifter
- Depth Anything V3 (ByteDance, 2025) — feed-forward 3DGS head we'd swap in for Hunyuan3D if available

---

Run it yourself

Just the demo (no GPU, no pipeline)

Open https://downtoearth-9lq.pages.dev — it runs entirely in the browser.

To serve locally:

```
cd voxgaussian\viewer\public
python -m http.server 5174
# open http://localhost:5174/uvw_demo.html
```

The Python module

```
pip install numpy pillow
python -m voxgaussian.pipeline.uvw_atlas
# expected output:
#   doctest: 8/8 passed
#   Exhaustive bijection check (256³ pairs)... OK
```

Project status: locked at iter 1 (with the Lyra-2-shaped scaffolding in place)

After exploring iterative refinement on a consumer-grade backbone (Juggernaut XL), we've landed on a deliberate stopping point: **the pipeline locks at `--max-iterations 1` by default and the deployed demo serves the iter-1 baked Gaussian cloud**. Bootstrap + one corroborating inpaint pass is the quality plateau on this hardware class — iter 2+ degrades the result even with all three of the Lyra-2-style architectural fixes wired in.

Architectural pieces that did ship and stay in the code:

Fix	Layer	What it does
Canonical-coord	inpaint conditioning	Renders the current voxel state with each pixel's RGB = its voxel coord, feeds it as a second ControlNet so the diffusion model is told "non-black pixels are

Fix	Layer	What it does
ControlNet anchor		locked geometry, only fill black holes." Wired via `workflows/depth_semantic_inpaint.json` node 31/61.
Fill-holes vote gating	propagate loop	`propagate.py` skips voting for pixels that the original render already saw cleanly — only the genuine disocclusion holes get written back. Stops histogram dilution.
50/50 found-vs-unfound view selection	camera picking	`select_view.py` probe-renders each candidate at 32^2 and scores by `sqrt(found × unfound)`. Geometric mean peaks when the frame has half-context, half-holes — the sweet spot for inpaint anchoring.
UVW bijection / atlas	data structure	The bidirectional voxel ↔ RGB packing is the foundation everything else stands on. See `pipeline/uvw_atlas.py` and the UVW explainer demo.

What we decided NOT to pursue (and why):

- **Cloning NVIDIA Lyra 2.0.** Research-only license; cannot be distributed, deployed, or sublicensed. Would also need an H100 / GB200 to run. Out of scope.
- **GEN3C-Cosmos-7B as an inpaint backend.** Realistic alternative with a usable license, similar pattern, ~RTX-class hardware. Not pursued in this round; documented as the natural Option B if someone wants to push past the current quality ceiling.
- **Continuous self-improve mode.** We had a WebSocket STOP button and `--continuous` flag wired up for "run until satisfied"; both the button (in the live viewer) and the codepath (`stop_requested` in `refine.py`) remain in place for graceful aborts, but the CLI flag is gone and iter 2+ runs are research-only.

The live demos at https://downtoearth-9lq.pages.dev are the canonical outputs. The 167k-Gaussian photoreal scene is an iter-1 bake. Rebuilding it from scratch:

```
cd voxgaussian
python -m pipeline.refine --scene hamlet-square    # default: max-iterations=1
# bootstrap (~3 min) → iter 1 inpaint (~1 min) → Phase B → exits
cp runs\hamlet-square\gaussians.json viewer\public\scene\hamlet-square.gaussians.json
git add voxgaussian/viewer/public/scene/ ; git commit -m "rebake hamlet" ; git push
npx wrangler pages deploy voxgaussian/viewer/public --project-name=downtoearth --branch=main
```

That's the production path — single iter, deterministic, ships.

The full pipeline (local generation)

Prerequisites:

- **ComfyUI** running at `http://127.0.0.1:8188`
- Custom nodes: `ComfyUI-Hunyuan3DWrapper`, `ComfyUI-Trellis` (optional), `ComfyUI_IPAdapter_plus`
- Checkpoints: Juggernaut XL v9 (or later)
- Python deps: `pip install -r pipeline/requirements.txt`

Generate everything:

```
cd pipeline
python run_all.py
```

Or step-by-step:

```
python gen_scene.py                  # Juggernaut XL → scene PNGs
python scene_to_3d.py                # PNGs → 3D meshes (Hunyuan3D)
python extract_walkable.py           # Walkable polygons from depth + normals
python gen_character.py              # Front + back image + Hunyuan3D MV → character GLB
```

Then start the live viewer:

```
cd ..\voxgaussian
python -m pipeline.refine --scene hamlet-square
# open http://localhost:5174 — click UVW button to see the atlas in action
```

Quest 3 deployment

Local dev: visit `http://<dev-machine-LAN-IP>:5174` from Meta Browser. Click **ENTER VR** for immersive mode or **ENTER PASSTHROUGH** for AR.

Production: host `voxgaussian/viewer/public/` on any HTTPS static host (Cloudflare Pages, Netlify, GitHub Pages). WebXR needs HTTPS off localhost.

Folder layout

```
DownToEarth/
├── README.md                        (this file)
├── LICENSE                          (MIT)
├── pipeline/                        (parent walker — Juggernaut → Hunyuan3D → walkable polys
│   ├── gen_scene.py
│   ├── scene_to_3d.py
│   ├── extract_walkable.py
│   ├── gen_character.py
│   ├── run_all.py
│   └── workflows/                   (ComfyUI workflow JSON templates)
├── viewer/                          (parent walker viewer — Three.js + WebXR)
│   ├── server.js
│   └── public/
│       ├── index.html
│       └── js/{app.js, scene-loader.js, walker.js, dialog.js}
└── voxgaussian/                     (this sub-project — iterative voxel refinement + UVW atl
    ├── README.md                    (design memo)
    ├── pipeline/
    │   ├── uvw_atlas.py             ★ the bijection + summary atlas builder
```

```
│   ├── voxel_store.py          sparse per-voxel class histograms
│   ├── bootstrap.py            init from Mapillary + OneFormer + DA-V2
│   ├── render_voxels.py        voxel → depth+semantic image
│   ├── select_view.py          active view selection
│   ├── propagate.py            vote propagation + ray-carving
│   ├── inpaint_client.py       ComfyUI ControlNet client
│   ├── refine.py               main refinement loop
│   ├── texture_pass.py         Phase B colors
│   ├── gaussian_fit.py         Phase B splat fitting
│   └── live_server.py          WebSocket bridge + HTTP static
├── viewer/public/
│   ├── index.html              ★ live pipeline-driven viewer (UVW mode)
│   ├── uvw_demo.html           ★ standalone interactive demo
│   ├── _redirects              Cloudflare Pages: / → /uvw_demo
│   └── js/app.js               ★ UVW integration
├── runs/                       per-scene refinement output (gitignored)
└── workflows/                  ComfyUI inpaint workflows
```

← SCENE DEMO UVW EXPLAINER SOURCE ON GITHUB

Files marked ★ are the UVW atlas surfaces — Python implementation, standalone WebGL demo, and live viewer integration respectively.

License

MIT. Original work by MiLO with collaborative design from Claude (Opus 4.7). No NVIDIA Lyra code, no GPL dependencies. Ship freely.