IST Vector

Architecture & Developer Guide

Companion to the Reference Manual — Issued June 2026

What this guide is

Subject	The internal structure of the IST Vector Postprocessor: how the source tree is laid out, how a mesh travels from file to pixels, and where to cut in when you extend it.
Audience	Engineers reading or modifying the C# code, and integrators embedding the `ISTVectorPostprocessor.Core` library. The Reference Manual is the *user's* view of the same system; this is the *implementer's*.
Organisation	§1 is the solution layout and build. §2 is the data model (the one type everything orbits). §3 walks the load-to-render pipeline. §4 is the `Core` subsystems, §5 the desktop application. §6 is the concurrency and caching model. §7 covers Native AOT and cross-platform constraints. §8 is the test strategy. §9 is a set of "how to extend" recipes. §10 is the repository map.
What it is not	A user tutorial (see the Reference Manual), nor an API reference dump — `Core` ships with `GenerateDocumentationFile=false` and is not a public NuGet surface. Names below are stable enough to navigate by; line numbers are deliberately omitted because they rot.

1. Solution layout and build

The product is one .NET solution (`csharp/ISTVectorPostprocessor.sln`) of six projects — the four below plus the `tools/IconForge` icon renderer and the `installer/windows/IlScrambler` IL-renaming tool — alongside a cross-platform installer-script tree. The load-bearing rule is the dependency direction:

Invariant. Core is headless	`ISTVectorPostprocessor.Core` has *no* dependency on Avalonia, OpenGL, or any windowing system — only the BCL and `System.Numerics`. Every parser, the recovery mathematics, the scene builder, the software rasteriser, and the PDF writer live here and run without a display. The GPU and the UI live exclusively in `Desktop`. This is why the whole pipeline is unit-testable on a headless CI runner.

Project	Output	Role
`ISTVectorPostprocessor.Core`	**library**	Pure compute: models, parsers, geometry, recovery, scene building, raster + PDF export, embedded demos. Depends only on the BCL.
`ISTVectorPostprocessor.Desktop`	**exe**	Avalonia UI, the live OpenGL renderer, video writers, per-user persistence. References Core. The shipping artifact.
`ISTVectorPostprocessor.Core.Tests`	**xUnit**	Pipeline tests against a real solver-output corpus plus synthetic and adversarial fixtures.
`ISTVectorPostprocessor.Desktop.Tests`	**xUnit**	UI-adjacent and pure-helper tests (headless Avalonia; codec-dependent tests self-skip).

Outside the solution: `tools/IconForge/` renders the toolbar icon set; `installer/` holds the Windows and Linux packaging pipelines (§7).

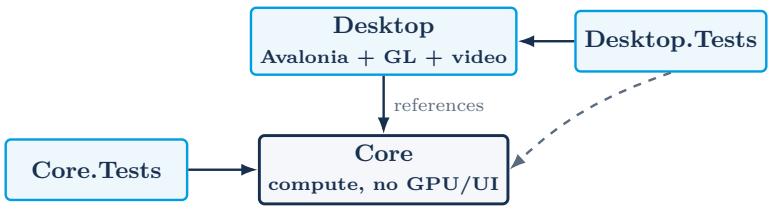

Figure 1. Project graph. Arrows point from dependent to dependency. `Core` sits at the bottom of the graph with no UI/GPU edges, which is what keeps the entire pipeline runnable — and tested — without a display.

1.1 Toolchain and targets

Item	Value
Target framework	`net10.0`, C# `latest`, `Nullable=enable`, `ImplicitUsings=enable`.
UI stack	Avalonia 11.3.7 with Semi.Avalonia + FluentAvalonia; SkiaSharp backend; `Avalonia.Svg.Skia` pinned 11.2.7 to hold the SkiaSharp graph at 2.88.9.
Release model	NativeAOT, self-contained, `TrimMode=full`, server GC, tiered PGO. There is no JIT / single-file fallback in the shipping path (§7).
Runtime IDs	`win-x64`, `win-arm64`, `linux-x64`, `linux-arm64`. macOS RIDs are intentionally absent (no `.app`/notarisation pipeline).
Metadata	`Directory.Build.props` is the single source of truth for Company / Authors / Product / Copyright / Version.

How to. Build and test	`dotnet build csharp/ISTVectorPostprocessor.sln` for a debug build; `dotnet test` runs the full suite (§8). A release artifact is a NativeAOT publish: `dotnet publish csharp/ISTVectorPostprocessor.Desktop -c Release -r win-x64`. The installer scripts under `installer/` wrap publish + packaging and auto-bootstrap the SDK / Inno Setup / appimagetool if absent.

How to. Build the manuals	The three LaTeX volumes (this guide, the Reference Manual, Inputs & Outputs) live at the repo root and build with `pdflatex` (run twice for cross-references). The Desktop `.csproj` carries a `BuildUserGuide` MSBuild target that regenerates the Reference Manual PDF on change and ships it beside the EXE for the in-app `Help` button.

2. The data model

Everything orbits one class: `ISTVectorPostprocessor.Core.Models.MeshData`. A parser fills it; the scene builder reads it; the renderers never see a file. Get this type and the rest of the code reads itself.

2.1 MeshData — the hub

Coordinates are stored **structure-of-arrays** (SoA), not as an array of points:

- `double[] NodesX, NodesY, NodesZ` — one column per axis (private setters; mutated through `SetNodesFlat`). `NodeCount` is `NodesX.Length`. SoA is deliberate: the hot paths (boundary extraction, iso-surface marching, the rasteriser) sweep one component at a time, so columnar layout keeps the cache line full and lets the JIT vectorise.
- `List<Element> Elements` — each `Element` is an `ElementType` plus a zero-based `int[] Conn`. A `Conn` entry of `-1` is the "no node" sentinel (emitted when an input connectivity index is non-positive) so a variable-data parser can still advance the right number of floats. **Every consumer that indexes coordinates by `Conn[k]` must guard with the canonical `(uint)idx >= (uint)nNodes`** — one branch that rejects both the negative sentinel and out-of-range positives.
- `Dictionary<string,FieldData> Fields` — the field table, keyed by display name; holds both the solver's literal fields and everything derived at load.
- `List<PartInfo> Parts` — per-part node/element ranges for multi-part meshes, including per-element-type offsets so a per-type variable block maps to the right global slice.
- `MeshOptimizationTier OptimizationTier` — the size tier (Standard → Huge) that gates eager derivation and precompute (Reference Manual §2.3).
- Transient (multi-step) storage and an optional `Action<int>? LazyStepLoader` callback that fills field arrays on demand when the step changes, so a multi-gigabyte transient need not be resident.

2.2 FieldData — one field, two layouts, N steps

`FieldData` carries a single scalar or vector field. Scalars are `float[]`; vectors are SoA `float[] X/Y/Z`. Three implementation features matter to anyone touching it:

- **Per-step storage** for transients, plus interpolation helpers (linear and Catmull–Rom) consumed by 60 Hz playback.
- **Ping-pong scratch** — a two-slot scratch buffer alternated per tick, so the GL thread's in-flight read from slot N never races the next tick's write to slot $N \oplus 1$, and playback allocates nothing.

- **Memoised range cache** — min/max is computed once per array identity; the second visit to the same array is $O(1)$. This is why a field switch is cheap on its second visit.

A `FieldKind` enum classifies provenance so the UI groups the combo boxes and so derivation does not recurse on its own output:

FieldKind	#	Origin
`Scalar`	0	Solver per-node / per-element scalar.
`TensorInvariant`	1	Derived Frobenius / von Mises / hydrostatic.
`VectorComponent`	2	One Cartesian component of a vector field.
`TensorComponent`	3	One matrix component of a tensor.
`VectorNorm`	4	Euclidean (ℓ^2) or uniform (ℓ^∞) norm.
`ErrorEstimator`	5	$\lvert f_{\text{nodal}} - \text{proj}(f_{\text{elem}})\rvert$ magnitude.
`Envelope`	6	Per-node min / max / range over all steps.
`MisalignmentAngle`	7	Angle between a material direction and the local principal-stress direction.
`Vector`	10	Solver vector field.
`TensorColumnVector`	11	Column vector assembled from tensor rows.

3. The pipeline: file to pixels

A file becomes a figure in four stages. The first three live in `Core`; the fourth has two sinks in `Core` (raster, PDF) and a third in `Desktop` (live GL).

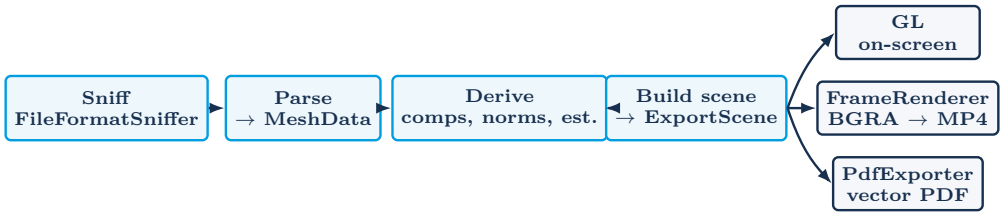

Figure 2. The pipeline. One `ExportScene` feeds all three render sinks — the implementation of the manual's "one scene, four outputs" promise (the fourth output being the saved-view JSON, which snapshots the state that *produces* the scene).

3.1 Sniff and parse

`Parsers.FileFormatSniffer.Detect(path)` reads the first few KB and returns a `DetectedFormat` (VTK legacy ASCII/binary; the five VTK XML variants; PVD; EnSight Case / Gold geometry ASCII & binary; OBJ; STL). Detection is by content, so a mislabelled extension still routes correctly; `DetectFromBytes` is the pure, testable core. The detected format selects a parser:

Parser	Handles
`CaseParser / CaseBinaryParser`	EnSight Gold Case: multi-part, transient (wildcard / arithmetic time sets), symmetric & asymmetric tensors, ghost cells, iblanking, per-axis coordinates. Fault-tolerant by design.
`VtkLegacyParser`	VTK legacy ASCII & binary; UnstructuredGrid, StructuredPoints, RectilinearGrid.
`VtkXmlReader + VtkXmlMeshBuilder`	XML plumbing (format / compression / byte order) feeding a piece-by-piece builder for .vtu / .vtp / .vti / .vtr / .vts.
`VtuParser / VtpParser`	UnstructuredGrid / PolyData specialisations.
`PvdParser`	ParaView collection (time-series manifest of .vtu / .vtp).
`StlParser / ObjParser`	Geometry-only meshes (no fields).

3.2 Derive at load

After parse, `Core.Rendering` populates derived fields: `VectorComponents` and `VectorNorms` for every vector, `TensorNorms` (Frobenius, von Mises, hydrostatic, column vectors) for every tensor, and `ErrorEstimators` where a nodal/element pair is detected by longest common prefix. `TensorInvariants` carries the Voigt-order tensor math. All of this is gated by `OptimizationTier`: at *Large* and above it is deferred to an explicit force-derive action to bound memory. Recovery (`FieldFilter.ApplyScalar` / `ApplyVector`: SPR, Z^2, MLS, Taubin, Perona–Malik) is *not* part of load — it runs on demand and rewrites the active field upstream of every representation.

3.3 Build scene, then render

`Rendering.SceneBuilder.Build(...)` is the heart of `Core`: it projects the (optionally deformed, optionally symmetry-expanded) 3-D mesh and its active field into a 2-D `Models.ExportScene` — a flat record of projected faces, edges, contour polylines, vector glyphs, tensor ellipses, trajectories, labels, dimensions, the colour bar, and the axis triad. Boundary faces come from `Geometry.BoundaryExtractor`; iso-geometry from `IsoSurfaceExtractor` / the `IsoLines` pipeline; per-vertex colour from `ColorScale` + `Colormap.TurboColormap`; shading from `LightRig` (Lambert + GGX). The scene is depth-sorted back-to-front so the consumers need no z-buffer.

Two `Core` sinks consume `ExportScene`: `Rendering.FrameRenderer.Render(scene)` produces a BGRA pixel buffer (Gouraud fill, Wu anti-aliased lines, painter's algorithm) for video and stills, and `Export.PdfExporter.Export(scene)` emits a vector PDF (Type-4 free-form Gouraud meshes, stroked polylines, base-14 Helvetica-Bold). The live on-screen sink is the `Desktop` GL renderer (§5). Because all three read the same scene, the figure on screen, in the MP4, and in the PDF are the same figure.

4. Core subsystems

Namespace	Files	What lives there
`Models`	9	`MeshData`, `FieldData`, `Element`, `PartInfo`, `ExportScene`, `CameraParams`, symmetry / dimension records.
`Parsers`	14	The format sniffer, every loader, and `TensorInvariants`.
`Geometry`	11	`BoundaryExtractor`, `FeatureEdgeDetector`, `IsoSurfaceExtractor`, `MeshQualityMetrics`, `PolygonClipper`, `FaceTable`, `Vec2`/`Vec3`.
`Geometry.IsoLines`	5	Iso-line extraction → polyline assembly → smoothing → label placement.
`Rendering`	33	`SceneBuilder`, `Camera`, `ColorScale`, `FieldFilter`, derived-field math, `LightRig`, `FrameRenderer`, `StressTrajectories`, `TensorEllipsoid`, view finders, overlay layout.
`Rendering.Gl`	2	`GlMeshRenderer` vertex format + GPU upload helpers and low-level `GlBindings` (the only GL-aware code in Core).
`Export`	2	`PdfExporter` (Type-4 mesh PDF) + helpers.
`Colormap`	1	`TurboColormap` — degree-6 polynomial Turbo, 256-entry LUT, NaN → magenta.
`Demos`	9	Procedural generators + embedded (gzip / binary) demo datasets and their loaders.

Two subsystems repay a closer read. **Iso-lines** are a four-stage pipeline (extract `IsoEdge` per crossing → `PolylineAssembler` stitches via spatial hash → `PolylineSmoother` optional Catmull–Rom → `IsoLineLabelPlacer` greedy multi-criteria placement), which is why each stage is independently testable. **Stress trajectories** (`Rendering.StressTrajectories`, namespace `ISTVectorPostprocessor.StressTrajectories`) implement an adaptive Dormand–Prince RK5(4) integrator over principal-stress eigenvectors.

5. The desktop application

5.1 Startup

`Program.Main` (the `[STAThread]` entry point) does four things before Avalonia starts: (1) captures a mesh path passed on the command line (so a launcher can open a case directly); (2) runs a **CPU preflight** on Windows — `Sse42`/`Popcnt` probed via `System.Runtime.Intrinsics`, refusing pre-2010 hardware with a clean `user32 MessageBoxW` rather than an opaque illegal-instruction crash; (3) configures Linux OpenGL (Mesa driver selection, optional `-software` fallback); (4) installs diagnostic logging and a top-level exception trap that surfaces a fatal dialog and writes the log. The Win32 message box is reached by P/Invoke precisely so it works when Avalonia itself is the thing that failed.

5.2 MVVM

`ViewModels.MainViewModel` is the application's state machine — mesh, active fields, time step, camera, display mode, contour / vector / tensor / deformation / symmetry / threshold / section / iso state, saved views. It is split across partials (`SceneBuild`, `ViewsAndPersistence`, `ViewFraming`) by concern. `Views.MainWindow` is likewise partialled (`FileIO`, `Camera`, `Playback`, `Export`, `Pdf`). Saved views and projects are plain data classes: `SavedView` / `ViewSnapshot` (the reproducible state) and the `sealed FigureRecipe` (the shareable `.spp` payload: schema version, app name, timestamp, mesh reference, snapshot).

5.3 The live renderer and controls

Control	Responsibility
`MeshGlSurface`	The interactive OpenGL renderer. All GL calls on the UI thread. Adaptive half-resolution FBO during interaction, full resolution on idle; shader-side displacement and contour-ribbon expansion so slider scrubs and rotation don't rebuild VBOs; generation-keyed cache rebuilds (§6).
`MeshViewport`	Wraps `MeshGlSurface`; paints the colour bar, axis triad, and info string on top.
`MeshOverlay`	Skia text overlay: iso labels, probe readout, contour annotations.
`GlDiagnostics`	Detects software GL (LLVMpipe / swrast) and surfaces recovery hints.

5.4 Video export and persistence

Video goes through `Export.FfmpegVideoWriter` (libx264, preferred) or `MediaFoundationVideoWriter` (Windows H.264 fallback); the Media Foundation file is compiled only for Windows RIDs, with a `.NonWindows` stub elsewhere — and the gate is on the *target* RID, not the build host, so a Linux runner cross-publishing `win-x64` still includes it. The optional `CinematicPost` pass adds FXAA / bloom / tone-map / vignette / grain per frame.

Per-user state lives under `PersistencePaths.Root` — `%APPDATA%\IST-VectorPostprocessor\` on Windows, `$HOME/.config/IST-VectorPostprocessor/` on Unix (via `Environment.SpecialFolder.ApplicationData`; a `UserProfile` fallback latches if that resolves empty). `ViewPreferenceStore`, `PartSessionStore`, `ProjectFileStore` (`.spp`), and `ExportFolderStore` all write through the same path provider, source-generated JSON, and atomic temp-then-rename (`Utilities.FileLocker` guards concurrent writers). The diagnostic log is per-machine under `%LocalAppData%`.

6. Concurrency, caching, and the thread model

The renderer reads `MeshData` on the GL/UI thread while playback and lazy loaders mutate it elsewhere. Four mechanisms keep that safe and cheap, and any change near them must preserve their contracts:

- **Generation counters.** `MeshData` holds `volatile int` fields bumped through `Interlocked.Increment`: `FieldGeneration` and `GeometryGeneration`. A step swap or field edit bumps the counter; GPU caches keyed on it rebuild automatically, with no explicit invalidation call.
- **Cache by (mesh, generation).** Boundary faces and other derived geometry are memoised in a `ConditionalWeakTable` keyed on the mesh, with the generation as the validity stamp — so the cache neither leaks (weak keys) nor goes stale (generation mismatch forces a rebuild).
- **Ping-pong scratch.** Per-tick scalar / vector interpolation alternates two scratch slots so an in-flight read never races the next write, and playback is allocation-free (§2.2).
- **Identity-keyed memoisation.** Field min/max and other reductions cache on array reference identity; the source-key gate short-circuits a 60 Hz refresh when the underlying arrays have not changed.

Invariant. GL stays on the UI thread	Every OpenGL call originates on the Avalonia UI thread. Background work (parsing, recovery, iso extraction, scene building) is pure `Core` computation that hands finished arrays back; it never touches GL. `CameraParams` is read as an atomic snapshot so a mid-frame camera edit can't tear. Parallel loops (parsers, `IsoSurfaceExtractor` with thread-local dedup, the uniform-grid recovery traversal) are confined to `Core` and join before returning.

7. Native AOT and cross-platform constraints

The shipping build is NativeAOT, full-trim, self-contained — there is no JIT fallback. That single decision propagates into rules every contributor must follow, because the AOT analyser turns most violations into build errors rather than runtime surprises:

Constraint	Consequence for contributors
No reflection-bound XAML	Compiled bindings are on by default; every `Window` / `DataTemplate` needs an `x:DataType`. A reflection binding trips IL2026 and renders empty.
Source-generated JSON	Every persistence type goes through a `JsonSerializerContext`; the reflection serializer is disabled, so a `JsonSerializer.Serialize(value)` without a `TypeInfo` throws loudly at the first call instead of silently regressing AOT.
Trimmer roots	The app assembly and the theme assemblies are rooted so the `avares://` resource manifest survives full trim; otherwise every embedded icon throws `FileNotFoundException` at startup.
Target-, not host-gating	Platform-specific package / source includes (e.g. Media Foundation, the Windows app manifest) condition on the *RID*, so cross-publishing `win-x64` from Linux produces a correct Windows binary.
Globalisation	`InvariantGlobalization=true` with `PredefinedCulturesOnly=false` (the theme constructs a non-invariant culture at startup; without the second flag it would crash).
CPU baseline	x86-64-v2 (SSE 4.2 + POPCNT), exactly what the §5.1 preflight gates on — any CPU that passes the preflight can run every AOT-emitted instruction.

Packaging lives in `installer/`: `windows/build.ps1` (Inno Setup, optional EV / Azure Trusted Signing, ConfuserEx), `linux/build.sh` (AppImage + tarball, optional GPG), `linux/build-deb-rpm.sh` (nfpm dual emission), and `publish-github.sh,ps1` for releases. `installer/README.md`, `CODE_SIGNING.md`, and `INSTALL_TROUBLESHOOTING.md` document the build, signing, and end-user warning paths in depth.

8. Test strategy

The suite is roughly **858 test methods (779 `[Fact]` + 79 `[Theory]`) across 124 files** (well over 1,700 executed cases once parameterised `[Theory]` rows are counted). It is organised by pipeline stage, and its design goal is that a regression cannot land silently.

Category	What it pins
Parsers	A real solver-output corpus (EnSight Case ASCII/binary, VTK legacy / XML, OBJ, STL under `TestData/`) plus fuzz and "brutality" fixtures that feed truncated, mislabelled, and non-finite input.
Geometry / Rendering	Boundary and feature-edge extraction, iso-lines / iso-surfaces, mesh quality, colour scale, derived fields, LIC, tensor glyphs, stress trajectories, scientific formatting.
Models	`MeshData` step swaps, transient interpolation, validation.
Regression	Named one-shot repros (each file is a fixed bug) so a previously-fixed defect re-breaking the build is caught by name.
Export / Demos	PDF exporter edge cases and recenter / squeeze transforms; demo bake & cache loaders.

Determinism is what makes regression baselines possible: the painter's-algorithm sort, the deterministic LIC noise lattice, and the byte-stable sub-triangle corners all produce bit-reproducible output across threads and runs. The GitHub Actions workflow `.github/workflows/dotnet-desktop.yml` builds and runs the suite on every push / PR; codec-dependent tests self-skip where FFmpeg / Media Foundation is absent, and Avalonia runs headless.

9. Extending IST Vector

The architecture concentrates extension points. Each recipe below names the seam and the one invariant you must not break.

How to. Add a file format	Write a parser that fills a `MeshData` (nodes SoA, `Elements`, `Fields`, `Parts`); teach `FileFormatSniffer.DetectFromBytes` the magic bytes (return a new `DetectedFormat`); add the dispatch arm. Fields land in the `Fields` dictionary; derived fields and the whole render path then come for free. Add a corpus fixture under `TestData/` and a fuzz test.

How to. Add a colormap	Mirror `Colormap.TurboColormap`: a 256-entry LUT sampled by index, NaN mapped to a sentinel colour. Wire it where `ColorScale` resolves the active map so every consumer (bar, surface, iso, PDF) shares it.
How to. Add a recovery filter	Add the kernel to `Rendering.FieldFilter` (`ApplyScalar` / `ApplyVector` dispatch on the integer `mode`) reusing the shared uniform-grid neighbour traversal and the NaN/$\pm\infty$ skip; expose it on the Field Filter card. It then governs every downstream representation automatically, because recovery is upstream of selection.
How to. Add a representation	Emit your primitive into `ExportScene` inside `SceneBuilder`, and make sure *both* `FrameRenderer` and `PdfExporter` render it. Add a card in `Desktop`. The non-negotiable below is what keeps screen, PDF, and MP4 identical.
How to. Add a derived field	Add a `FieldKind`, compute it in the derive-at-load path next to `VectorNorms` / `TensorNorms`, and respect `OptimizationTier` gating (skip at *Large*+ unless force-derived). Key the result so derivation never recurses on its own output.
Invariant. One scene, four outputs	Anything visible must travel through `ExportScene`. If a new primitive is drawn directly in `MeshGlSurface` but not emitted into the scene, it appears on screen and vanishes from the PDF and the MP4 — the single most common way to break the product's core promise. Render it once, into the scene; let all sinks consume it.

10. Repository map

Path	Contents
`csharp/ISTVectorPostprocessor.Core/`	The headless engine (§2–§4).
`csharp/ISTVectorPostprocessor.Desktop/`	The Avalonia app, GL renderer, video, persistence (§5).
`csharp/*.Tests/`	The xUnit suites and their `TestData/` corpus (§8).
`csharp/Directory.Build.props`	Shared assembly metadata (single source of truth).
`tools/IconForge/`	Toolbar icon renderer.
`installer/`	Windows + Linux packaging, signing, release scripts (§7).
`Assets/`	Icons (phone / laptop / bigscreen tiers), SVGs, the IST logo.
`IST_Vector_Report.tex`	The Reference Manual (the user's view).
`IST_Vector_InputsOutputs.tex`	The I/O contract (what is read, what is written).
`IST_Vector_Architecture.tex`	This guide.
`Audits/, AUDIT*.md`	Historical code-audit trails (file:line findings by severity).

IST Vector Postprocessor — Architecture & Developer Guide — P. Areias — pedro.areias@tecnico.ulisboa.pt — Instituto Superior Técnico, Universidade de Lisboa.