



An Interactive Three-Dimensional Volume Visualization Pipeline for Fusion Simulation Data in Unreal Engine and Omniverse

Darian Figuera-Michal , Eui-Hyun Jung , Mintae Na , Sun-Jeong Kim , Jong-Hyun Kim , Jung Lee , Yu-Seop Kim , Jungpyo Lee , Jaemin Seo , Il-Hwan Kim , Taeuk Moon , Jae-Min Kwon & Eisung Yoon

To cite this article: Darian Figuera-Michal , Eui-Hyun Jung , Mintae Na , Sun-Jeong Kim , Jong-Hyun Kim , Jung Lee , Yu-Seop Kim , Jungpyo Lee , Jaemin Seo , Il-Hwan Kim , Taeuk Moon , Jae-Min Kwon & Eisung Yoon (08 Sep 2026): An Interactive Three-Dimensional Volume Visualization Pipeline for Fusion Simulation Data in Unreal Engine and Omniverse, Fusion Science and Technology, DOI: [10.1080/15361055.2026.2721939](https://doi.org/10.1080/15361055.2026.2721939)

To link to this article: <https://doi.org/10.1080/15361055.2026.2721939>



© 2026 The Author(s). Published with license by Taylor & Francis Group, LLC.



Published online: 08 Sep 2026.



Submit your article to this journal [↗](#)



View related articles [↗](#)



View Crossmark data [↗](#)



An Interactive Three-Dimensional Volume Visualization Pipeline for Fusion Simulation Data in Unreal Engine and Omniverse

Darian Figuera-Michal^{a*}, Eui-Hyun Jung^{a*}, Mintae Na^b, Sun-Jeong Kim^b, Jong-Hyun Kim^c, Jung Lee^d, Yu-Seop Kim^b, Jungpyo Lee^e, Jaemin Seo^f, Il-Hwan Kim^f, Taeuk Moon^a, Jae-Min Kwon^g, and Eisung Yoon^g

^aDepartment of Nuclear Engineering, Ulsan National Institute of Science and Technology (UNIST), Ulsan, Republic of Korea; ^bDepartment of Convergence Software, Hallym University, Chuncheon, Republic of Korea; ^cDepartment of Design Technology, College of Software and Convergence, Inha University, Incheon, Republic of Korea; ^dDepartment of Computer Engineering, Hanbat National University, Daejeon, Republic of Korea; ^eDepartment of Nuclear Engineering, Hanyang University, Seoul, Republic of Korea; ^fDepartment of Physics, Chung-Ang University, Seoul, Republic of Korea; ^gFusion AI Research Center, Korea Institute of Fusion Energy (KFE), Daejeon, Republic of Korea

Received May 18, 2026

Accepted for Publication August 4, 2026

Abstract — We present an end-to-end pipeline that delivers gyrokinetic scalar fields into Unreal Engine and NVIDIA Omniverse, with three key advances. The first is the render time, native coordinate sampling of curvilinear structured grid fields, as exemplified by gKPSP on (r, θ, ϕ) , in both real-time engines, removing the intermediate Cartesian resampling pass. The second is the joint use of OpenVDB and OpenUSD as an engine-neutral intermediate representation between the solver output and the two engine front ends. The third is an interactive in-engine inspection through threshold window filtering, clip box dissolution, transfer function tuning, and arbitrary cross-section planes, with the native coordinate scene sustaining interactive rendering on a single graphics processing unit workstation.

The enabling mechanism is an inverse engine XYZ-to-toroidal map evaluated at every ray sample, returning each sample to the simulation frame before the structured grid lookup. The same map was implemented independently in the material, scientific renderer, and compute kernel layers of the two engines. A quantitative comparison against the Cartesian bounding box baselines showed that the native coordinate representation preserves the structured grid resolution and fine-scale turbulence content, which Cartesian resampling discards, at the cost of a bounded per-sample coordinate conversion overhead. The rendering of fields defined on unstructured high-order meshes is left to future work.

Keywords — Fusion digital twin, virtual tokamak, structured grid coordinate transform, OpenVDB, volume rendering.

I. INTRODUCTION

High-fidelity fusion simulations now routinely produce three-dimensional (3D) mesh and field data whose

interpretation strains the conventional toolbox of two-dimensional (2D) slices, contour overlays, and offline-rendered images. As whole-device modeling efforts move toward gyroradius-resolved transport on toroidal meshes, the data are neither modest in volume nor naturally Cartesian. Physically meaningful structures are spread across the geometry, coordinates, and time in ways that isolated post-processing views can obscure [1,2].

The pressure to interpret such data inside an interactive geometric context, rather than only through detached plots, has grown, together with the wider effort to build digital-twin environments around fusion devices in which the simulation results, engineering geometry,

CONTACT Eisung Yoon  esyoon@unist.ac.kr

*These authors contributed equally to this work.

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. The terms on which this article has been published allow the posting of the Accepted Manuscript in a repository by the author(s) or with their consent.

and operational data are expected to share a common spatial representation.

This pressure is particularly visible in the Korean fusion digital-twin program. The Versatile Virtual platform for Integrated fusion simuLation and anaLysis (WILL) platform and the V-KSTAR [3D digital twin platform that replicates the KSTAR (Korea Superconducting Tokamak Advanced Research)] effort have organized multiple high-fidelity codes, engineering geometries, and reactor design tools around a shared mesh and data infrastructure that are intended to support both physics analyses and device-scale virtual-machine workflows [3,4]. Within this ecosystem, two families of solver-side data coexist.

Unstructured meshes, managed through the Parallel Unstructured Mesh Infrastructure (PUMI) [5], a distributed mesh database developed at Rensselaer Polytechnic Institute's (RPI's) Scientific Computation Research Center (SCOREC) with native binary container formats (`.smb` for the discrete mesh and `.dmg` for the underlying geometric model), carry the mesh lineage of gKPSP2 [6] and its successors, and serve as the connective tissue between physics simulations, engineering computer-aided designs (CADs), interactive visualizations, and downstream analyses.

Alongside them, the gyrokinetic code gKPSP produces field snapshots on a curvilinear structured grid. These snapshots are handled end to end in structured form and do not pass through the unstructured mesh infrastructure. Mesh and field data handling, therefore, behaves as a shared infrastructure problem rather than as a peripheral visualization concern, and a missing or fragile data path between the solvers and the interactive layer rapidly becomes a program-level bottleneck.

At the rendering layer, real-time engines and digital engineering platforms, such as Unreal Engine and NVIDIA Omniverse, have matured into credible substrates for scientific scenes. They support immersive navigation, multidisplay deployment, and the integration of CAD with simulation outputs in production-grade environments, including dome-based scientific visualization rooms [2,7,8] and large-scale virtual production stages [9].

Recent fusion-specific examples include immersive plasma data viewers built around large-scale simulations [10,11], CAD and simulation digital twin scenes for fusion power plant design [12], and earlier virtual reality/augmented reality studies for fusion design engineering [13]. Together they sit downstream of the kind of digital engineering activity already documented for tokamak operations [14].

Game engine native facilities, such as sparse volume textures (SVTs) and heterogeneous volume (HV) materials, provide the building blocks for interactive volumetric scenes inside these workflows [15,16]. What these platforms do not provide is a fusion-aware path from the solver-side mesh and fusion-specific scalar and vector field data to engine-side assets. Questions of coordinate convention, signed field representation, time-dependent playback, and the mapping between non-Cartesian meshes and engine volume formats remain on the user side. They form the core technical challenge addressed in this paper.

Against this background, we present a mesh-to-volume workflow that connects PUMI-based fusion meshes and high-fidelity field data to interactive visualization environments built on Unreal Engine, and we demonstrate the portability of the same data concept into NVIDIA Omniverse. The workflow is exercised on KSTAR/gKPSP-related mesh and electrostatic potential field data, which we treat as a representative example of the broader class of fusion-specific scalar fields targeted by this pipeline.

The same per-sample handling extends componentwise to the vector fields when they are represented as tuples of scalar components for downstream analysis. The central positive outcome is that the fields whose simulation already lives on a curvilinear structured grid, as is the case for gKPSP on (r, θ, φ) , can be rendered directly inside both engines in their native coordinate form through the render time inverse map detailed in Sec. III.C.

The volumetric assets reach the engines through OpenVDB and OpenUSD [17,18]. The Omniverse implementation extends the data path into four concrete rungs that climb from a Cartesian baseline to a render time evaluation of the inverse toroidal map on the simulation's native grid. High-order field interpretation of the original unstructured mesh, which would require either a custom high-level shading language (HLSL) pixel shader basis evaluator or a refinement of the engine-side mesh, is explicitly deferred and discussed as a future extension in Secs. II and V.

The contribution of this work is not the proposal of a new graphics algorithm but the fusion-specific integration of existing mesh, volume, and real-time rendering technologies into a documented digital twin-enabling visualization layer. We deliberately use digital twin-enabling rather than digital twin because the demonstrated path consumes solver output one way into an inspection scene, while closed-loop coupling with operational data and feedback to the simulator are out of scope here and reserved for future integration.

Three concrete elements distinguish the present integration from its closest precedents, notably the United Kingdom Atomic Energy Authority's (UKAEA's) STEP/Omniverse digital twin work [12] and the GEOMAR Kolumbo benchmark in Unreal Engine 5 [8]. The first is a material-side toroidal-to-Cartesian inverse map that lets a structured grid gyrokinetic field volume be sampled in its native simulation coordinates rather than as a pre-sampled Cartesian VDB. The second is the deliberate use of OpenVDB and OpenUSD jointly as an engine-neutral intermediate representation between Unreal Engine and Omniverse. The third is a dependency-light PUMI .smb ingestion path into Unreal Engine, implemented as a details panel–customized C++ loader that mirrors PUMI's binary import pattern in engine, and therefore does not require linking against the message passing interface (MPI)–oriented PUMI run time.

The remainder of the paper is organized around these three elements. Sec. II develops the data requirements that motivate them. Sec. III presents the approach itself as the three key technical developments of native coordinate sampling, engine-neutral representation, and ingestion. Sec. IV reports the demonstrations and quantitative results obtained in both engines, and Sec. V discusses the limitations and extensions. Engine-specific implementation and operating details are collected in [Appendix A](#) (Unreal Engine) and [Appendix B](#) (Omniverse).

II. FUSION MESH AND FIELD DATA REQUIREMENTS FOR DIGITAL TWIN VISUALIZATION

The target data for this work originated from high-fidelity fusion simulation workflows in which the mesh geometry, coordinate systems, and field quantities were tightly coupled rather than separable concerns. Two solver branches supplied the present demonstration, and they feed two separate data paths through this paper. The field volume branch carries structured grid field snapshots to the volume renderers, and the mesh-geometry branch carries unstructured mesh geometry to the engine-side loader.

The first solver is gKPSP [19], a bounce-kinetic particle simulation code that supplies the time-evolving electrostatic-potential snapshots used in Secs. IV.A and IV.B. gKPSP integrates the gyrokinetic equations on a structured radial-poloidal-toroidal grid in (r, θ, φ) . The resulting (r, θ, φ) triad is a curvilinear orthogonal coordinate system in the toroidal small aspect ratio limit, so the snapshot data inherit a curvilinear but regular and locally orthogonal indexing that is exploited explicitly by the workflow described in Sec. III.

The second solver is gKPSP2 [6], a four-dimensional gyrokinetic continuum solver based on the discontinuous Galerkin method whose PUMI-managed mesh is the geometry vehicle for the loader exercise. gKPSP2 assumes axisymmetry in the toroidal direction and resolves real space as a 2D (R, Z) poloidal plane, with two velocity-space dimensions completing the four-dimensional phase space. The PUMI mesh files describe this 2D poloidal-plane mesh directly, and the solver's discontinuous Galerkin field expansion lives on that mesh, so the mesh itself is part of the physics representation rather than a purely graphical scaffold for postprocessing.

The volumetric demonstrations in this paper bypass the unstructured mesh entirely. The gKPSP field snapshots flow through the field volume branch as structured (r, θ, φ) arrays and are never routed through the PUMI infrastructure, which appears only in the geometry-only ingestion exercise of the mesh-geometry branch (Sec. III.A). The same PUMI-managed geometry, distributed in PUMI's native .smb (binary mesh container) and .dmg (geometric model) files, also feeds related codes inside the Korean fusion digital twin program.

The natural successor solver gKPSP3 [20], a five-dimensional gyrokinetic continuum solver, additionally introduces curved, high-order, and toroidally segmented (wedge decomposed) 3D elements with nonconforming inter-wedge interfaces. Its visualization requires curved element rendering machinery beyond the scope of the present demonstration, and we therefore have restricted the demonstrated workflow to gKPSP-supplied field data and the gKPSP2-managed PUMI mesh, while flagging gKPSP3 as a future integration target.

A digital twin visualization layer for fusion data must therefore preserve the relationships among the geometry, field values, coordinate definitions, and time evolution at the same time. For toroidal plasma simulations, this requirement is more demanding than for typical visualization use cases because the data are not naturally stored in a simple Cartesian coordinate system. The underlying samples follow toroidal angles and field-aligned directions associated with the WILL-class data path [3], and they may carry signed scalar quantities, such as electrostatic potential, whose sign carries physical meaning.

Naively flattening such data into an axis-aligned Cartesian box, as common visualization toolchains assume, would inflate the storage footprint and risk losing the coordinate information that connects the rendered image back to the physics. The voxel budget and flux surface misalignment costs of this Cartesian intermediate are documented in Sec. III.C. A key design choice in the present workflow, therefore, was to keep the data on the

solver native grid and to absorb the engine-side coordinate convention into the rendering material rather than into the on disk asset.

For the requirements analysis that follows, it suffices that the demonstrated workflow must accept solver-side mesh artifacts in their original PUMI native form and field snapshots through their structured grid volume path, and must remain compatible with the V-KSTAR and WILL interactive visualization and reactor design environments [3,4]. The detailed implementation and assessment in the remainder of this paper are limited to the interactive visualization of the high-fidelity gyrokinetic field data. Deeper integration with the other codes of the Korean fusion digital twin program, enumerated in Sec. V, is left as future work.

A direct element-by-element implementation of such fusion meshes inside Unreal Engine is not scalable. Representing each mesh element as an individual engine actor, in the way one might prototype a small parametric mesh, would create a number of engine objects comparable to the number of elements in the simulation mesh, which for ITER-class gyrokinetic discretizations approaches $\mathcal{O}(10^8)$ vertices when the poloidal mesh is resolved at the ion gyroradius.

Engine-side scaling infrastructure, such as instanced static meshes, world partition, and the data-oriented mass entity ECS (entity component system) framework [21], all raise the practical ceiling on actor-like objects by orders of magnitude relative to a naive one actor per thing design. Even so, the simulation mesh element count remains several orders of magnitude beyond what an engine scene populated with that many tickable actors would handle interactively, while still leaving high-order field interpretation as an actor-side responsibility.

On the consolidated actor side, Unreal Engine offers Nanite, a virtualized geometry pipeline that lets a single static mesh carry essentially unbounded polygon counts behind a fixed run-time cost by streaming and culling micropolygon clusters [22]. Once a fusion mesh has been collapsed into one engine object, Nanite, therefore, is a natural substrate for keeping that mesh interactively renderable even at ITER-class polygon counts.

The workflow accordingly adopts the alternative strategy of carrying one actor with the whole mesh and performing field interpretation in the material rather than at the actor level, so that consolidated mesh or volume representations remain in a small number of engine objects while transfer function logic, coordinate handling, and shading move to the engine-side material, volume, or shader nodes.

A second alternative considered in the design phase was a fully unstructured ray sampling method that would

preserve the original mesh representation most directly. Two implementation obstacles stood in the way of this route. The first obstacle was per-sample point location. The approach required, at every ray sample, a fast query that determined which mesh element contained that sample so the element's basis functions could be evaluated on the fly. Implemented robustly on the graphics processing unit (GPU), this would in turn require dedicated acceleration structures integrated with the engine's ray marching and material pipeline [7,23]. Three established families illustrate the design space:

1. Bounding volume hierarchies routinely used in production CPU/GPU ray marching kernels [24,25].
2. Spatial hash tables recently deployed by the NuBDeC fast-ion collision-detection workflow on KSTAR-scale tokamak meshes [26].
3. Adjacency-based searches through neighboring elements, as exploited by the PUMI unstructured mesh particle-in-cell (PIC) library [27].

The second, more fundamental obstacle was that the engine's standard rasterizer path interpolates per-vertex material outputs linearly across each triangle. The vertex interpolator material expression evaluates its input once per vertex in the vertex shader, after which the values are interpolated barycentrically by the hardware rasterizer before reaching the pixel shader [28]. Faithfully reproducing the high-order discontinuous Galerkin basis function expansion used by gKPSP2 inside this pipeline, therefore, would require either bypassing the rasterizer interpolation through a fully custom HLSL pixel shader evaluation per element, or refining the rendered mesh to the point where the linear interpolation was dense enough to be a faithful approximation.

Both routes would amount to redesigning the engine's interpolation substrate rather than configuring it, and the per-sample arithmetic added by such a custom evaluator would in turn require dedicated code optimization on the GPU shader side to keep the rendered scene at interactive frame rates rather than at offline ones. While attractive for fidelity, building and maintaining either a GPU-side mesh search or a custom high-order interpolator, together with the shader-level performance work needed to keep the result interactive, was outside the practical scope of the present implementation, and was therefore deferred.

With element-level actors ruled out for scalability and unstructured ray sampling deferred for engineering

reasons, the visualization problem was reformulated as a conversion problem in which fusion simulation fields must be transformed into engine-compatible volume assets while retaining their physically meaningful coordinate and geometry relationships. This formulation, which defines a small, well-posed design space rather than an open-ended graphics question, led to two concrete implementation routes that are evaluated in the remainder of this paper.

The first route, which uses flipbook-style pseudo volume textures (PVTs) with ray marching in the engine material [23], is documented as the initial baseline. The second route uses OpenVDB [17,18] sparse volume files imported as SVTs and rendered with HV materials, following a recent survey of large-volume rendering in Unreal Engine [8].

The remainder of this paper presents these two routes as successive answers to the same data pathway question and uses the second route, on a structured grid gyrokinetic field, to characterize what a digital twin-enabling visualization layer for fusion data must provide.

III. APPROACH: THREE KEY TECHNICAL DEVELOPMENTS

The proposed workflow takes simulation-side mesh and field data as its input and decomposes the visualization process into a small set of file-level conversion stages, namely mesh import, field-to-volume conversion, volume asset creation, and digital twin scene integration, whose outputs are documented file artifacts. The two demonstrations exercised in this paper deliberately addressed only a subset of the staged pipeline.

On the mesh side, the present implementation established that PUMI .smb geometry can be ingested into Unreal Engine and rendered as a static mesh through the engine's StaticMesh asset path. Field interpolation on top of the unstructured mesh was verified only at the level of an analytical, per-vertex linear test field on a tetrahedron rather than as a coupled mesh field-rendering pipeline.

On the field side, the volumetric demonstrations bypassed the unstructured mesh entirely and operated on engine-side volume assets converted from gKPSP electrostatic potential snapshots. These assets were regularized OpenVDB volumes for the Unreal Engine route and the Omniverse Cartesian baselines, and a raw structured container for the Omniverse render time paths (Sec. IV.B). The device geometry was specified in the engine through a small number of manually entered parameters (notably the tokamak major and minor radii) that drove the material-side coordinate transform in Eq. (1),

$$\begin{aligned}\varphi &= \text{atan2}(y, x) , \\ R &= \sqrt{x^2 + y^2} , \\ r &= \sqrt{(R - R_0)^2 + z^2} , \\ \theta &= \text{atan2}(z, R - R_0) .\end{aligned}\tag{1}$$

The motivation for this staged structure was physical. Research codes for fusion plasmas typically advance on micro- to millisecond timescales, whereas a digital twin user inspecting the data needs frames produced at the millisecond-to-second timescale on which a human observer can perceive continuous motion. Isolating the solver-side and engine-side stages behind documented file artifacts is what allows each side to operate on its natural timescale without blocking the other.

The boundaries were concrete file formats, with a PUMI .smb file on the mesh side, an OpenVDB sequence on the field side, and an OpenUSD scene on the integration side (see Fig. 1). The same solver output, therefore, could serve both the Unreal Engine and Omniverse front ends without per-engine rebuilds, and the limitations could be traced to a single stage at a time rather than to the pipeline as a whole. Engine-specific implementation and operating details are collected in Appendix A (Unreal Engine) and Appendix B (Omniverse).

III.A. Dependency-Light PUMI Mesh Ingestion

The choice to feed Unreal Engine and Omniverse with PUMI mesh files in the first place was motivated by what PUMI brings to the data side of a fusion visualization stack. PUMI's internal data structure is engineered for high-performance computing handling of unstructured meshes. It exposes mesh entities (vertices, edges, faces, regions) and their adjacency queries as first-class operations, supports node-, edge-, face-, and element-centered fields and tags directly attached to the mesh, and ships shape function machinery for high-order elements on a range of element types [5].

The mesh description framework that ships with Unreal Engine and the equivalent OpenUSD UsdGeom primitives in Omniverse provide their own mesh handling application programming interfaces (APIs), but those APIs are aimed at rendering rather than at hosting solver-side fields on top of the mesh, and the adjacency queries and high-order field machinery that scientific computing relies on are not part of the engine-side mesh model.

The present work, therefore, first verified that a PUMI .smb mesh could be brought into the engine at all and rendered as an Unreal StaticMesh. Making

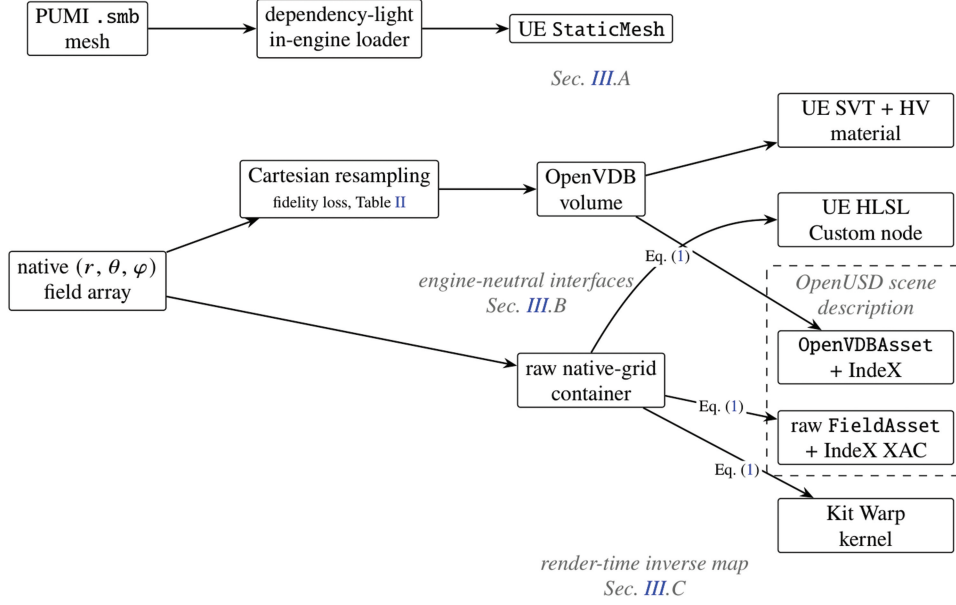


Fig. 1. Two-branch data path schematic of the proposed workflow. In the mesh geometry branch, the PUMI .smb mesh is parsed by the dependency-light in-engine loader and reconstructed as an Unreal engine StaticMesh (Sec. III.A). In the field volume branch, the native (r, θ, φ) field array reaches the renderers along two routes. The first is a conversion time Cartesian resampling into OpenVDB volumes consumed by the Unreal SVT/HV pipeline and by the UsdVol/Index pipeline, with the associated fidelity loss quantified in Table 2. The second is comprised of render time routes in which the raw native grid container is sampled by the Unreal HLSL Custom node, the Index Xtreme Accelerated Computing (XAC) program, and the Kit Warp kernel, each evaluating the inverse toroidal map of Eq. (1) per ray sample. Eq. (1) is evaluated only on these native render time paths. OpenUSD acts as the scene description layer spanning the Omniverse-side consumers.

PUMI’s field handling subset available in engine through a dedicated library is left as a follow-on direction.

The implementation reimplemented the parts of PUMI’s .smb reader required for geometry ingestion directly inside the engine module so the loader did not need to link against the upstream PUMI library or its MPI-oriented build dependencies. This kept the Unreal-side dependency footprint small and avoided dragging a MPI stack into the editor process. The parsed vertex, edge, and face arrays were fed into Unreal’s Mesh Description framework, from which a StaticMesh asset was constructed and instantiated as the PumiActor’s rendered geometry through a custom PumiStaticMeshComponent (the engine-side mesh component that binds the StaticMesh to the actor and exposes the mesh metadata to the editor) [5,29].

The route was exercised on a KSTAR equilibrium-derived poloidal mesh generated from a gEQDSK magnetic equilibrium reconstruction, with 40 110 vertices, 119 934 edges, and 79 825 triangles, a mesh dimensionality of $\text{Dim} = 2$, and a single part (Fig. 2). In the viewport rendering in Fig. 2, triangles whose winding order differed from the engine’s front face convention appear culled from some view directions under the default single-sided material. This patch-like appearance, which is a cosmetic

artifact of the ingestion path rather than missing geometry, disappears when a two-sided material is assigned.

Two further design choices in this loader deserve explicit comment. First, only the mesh geometry is currently ingested through this path, and node- or element-centered field data are not yet attached on the engine side. The present role of the mesh pathway, therefore, is to provide an existence proof that the solver-side fusion mesh geometry can enter the engine without manual rebuild rather than a fully featured field loader. The field-on-mesh interpolation was verified only at the level of a small parametric test on a single tetrahedral element. This demonstrated that the static mesh pathway can carry per-vertex scalar fields but not yet the high-order solver fields produced by gKPSP2.

Faithful high-order rendering would require the user to author shader code that places nodes and evaluates shape functions according to the underlying high-order basis because Unreal Engine itself does not support high-order elements (see Sec. II for why replacing the engine’s vertex-to-pixel interpolation substrate would be required). Second, the loader currently expects a single-part .smb file. In gyrokinetic practice, this is the natural input format. Even at ITER-class poloidal resolution, where a gyroradius-resolved mesh approaches $\mathcal{O}(10^8)$ vertices,

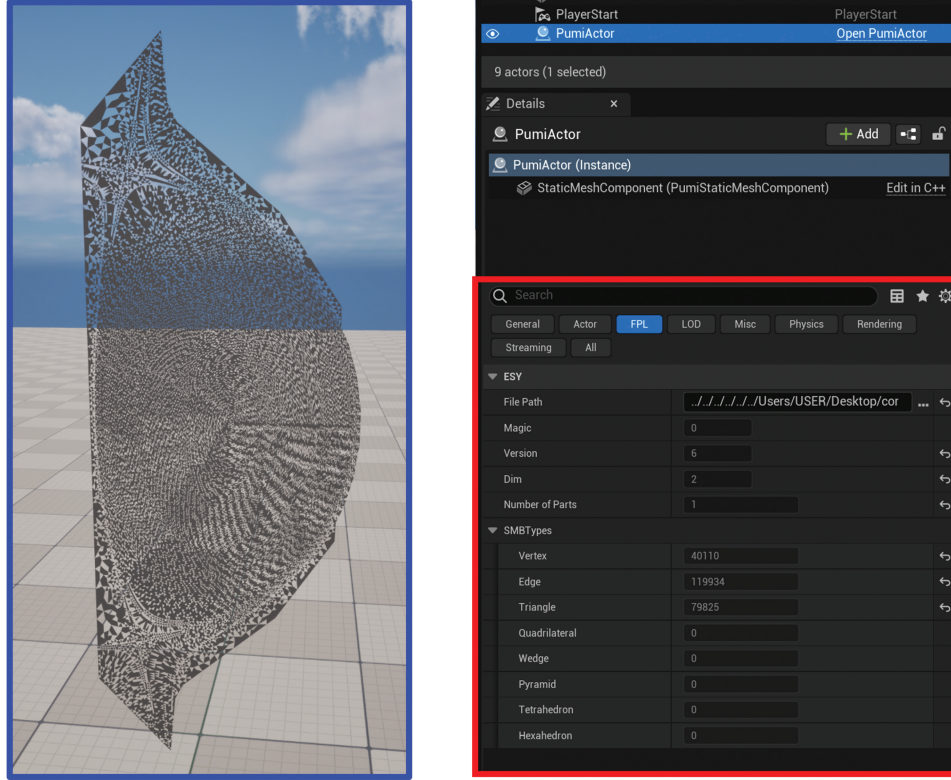


Fig. 2. KSTAR gEQDSK-based PUMI mesh imported via the in engine loader. The left panel shows the viewport with a half-poloidal cross section. The right panel shows the outliner listing the PumiActor with its PumiStaticMeshComponent, together with the details panel in which the FPL category is populated with the parsed PUMI header fields.

the global mesh is generated as a single .smb container before the solver partitions it at run time.

PUMI also provides utilities for merging an already-partitioned mesh back into a single file when required [5], so multi-part ingestion is listed as a deferred extension rather than as a blocking limitation for the demonstrated use case. The full loader internals and the editor-side workflow are documented in Sec. A.I in [Appendix A](#).

III.B. Engine-Neutral Field and Scene Interfaces

The engine-side field pathway went through two generations, and the first is summarized here only for the constraints it established. That first generation, which was a flipbook plus PVT route in which the snapshot atlases were ray marched by a custom material graph, confirmed that time-dependent fusion field data could be played back inside an engine scene.

Exercising it exposed three limitations that motivated the upgrade. First, the atlas sampling pattern reduced sampling between adjacent slices to a manually blended linear interpolation that mismatched the higher-order discontinuous Galerkin basis-function expansion of the

underlying gKPSP2 simulation. Second, streaming every snapshot into the engine through a run-time image-import step introduced a perceptible playback delay between the snapshots. Third, ray accumulation in the grayscale space blurred the boundary between the positive and negative regions of the signed potential field.

Together these limitations marked the shader graph PVT path as a workable demonstration substrate rather than a satisfactory representation of sparse, signed, 3D fusion fields. The atlas mechanics and the signed field RGB (red, green, blue)-PVT remedy are documented in Sec. A.II in [Appendix A](#).

The upgraded field pathway converts the same snapshots into OpenVDB volumes [17], whose GPU friendly variant NanoVDB [18] underlies the engine's sparse volume run time and imports them into Unreal Engine 5.6.1 as SVTs consumed by HV materials [15,16]. The selection of this route followed a survey of large-volume techniques in Unreal Engine 5, including the GEOMAR/Schlüter benchmark on the Kolumbo dense $4211 \times 1501 \times 935$ ocean floor data set in which a SVT-based HV pipeline was shown to scale into the multigigavoxel regime after appropriate engine-side overflow fixes [8].

This SVT/HV combination matched the present fusion use case for several reasons. The sparse tile budget aligned with the morphology of the plasma volume, in which most of the bounding box voxels lay outside the magnetically interesting region. The HV material exposed density, extinction, emission, and transfer function controls as named material parameters, which made it possible to render signed fields with explicit positive/negative color mapping rather than through accumulated grayscale [8]. Finally, the route exposed internal inspection through threshold filtering and material-driven clipping out of the box.

Across both engines, the field payload takes one of two engine-neutral forms, a FloatGrid OpenVDB volume for the conversion time Cartesian baselines and a raw floating-point native grid container for the render time paths. The scene wrapper differed by engine. Unreal Engine imports the OpenVDB sequence directly as a SVT asset bound to a HV material, while Omniverse binds the same class of asset as a `UsdVol.Volumeprim` with an `OpenVDBAsset` field reference. Resampling happens at conversion time for paths (a) and (b), and not at all for the render time paths (c) and (d), which keep the simulation array in its native form until each ray-march sample (see Table 1).

Each render time evaluator implements the same inverse map in its host’s shading or compute language, through a HLSL Custom node in the Unreal material, a XAC program in the Index backend, and a Warp kernel in the Kit extension. In this arrangement, OpenVDB and OpenUSD act as complementary layers rather than as a single joint intermediate representation. OpenVDB carries the field payload, while OpenUSD carries the scene description.

One portability bound of the raw container route belongs in this matrix. Its scene binding is an Omniverse/Index-specific extension of the OpenUSD `UsdVolVolume` field family rather than a portable schema. The USD stage, therefore, encodes the raw data binding, renderer selection, coordinate transform, normalization, and color mapping in one self-contained scene

description that is portable between the Omniverse/Index hosts. Reading the same scene on a USD renderer without Index support reverts to the base `UsdVolVolume` behavior and loses the XAC-driven sampling. The four concrete operating points realizing this matrix are summarized in Table 1 (Sec. IV).

III.C. Render Time Inverse Toroidal Map

A central component of the workflow is the treatment of the coordinate systems within the Unreal material logic, and it is here that the structured grid character of the gKPSP field becomes a positive asset rather than a complication. The original gKPSP field is defined on a structured radial-poloidal-toroidal grid, while the engine samples the SVT in its own object space Cartesian coordinates. For each sample at object-space position (x, y, z) , the HV material therefore evaluates the inverse map to toroidal coordinates using the two-argument arctangent (`atan2`), which resolves the quadrant of both the geometric toroidal angle φ and the poloidal angle θ [see Eq. (1)] inside a custom HLSL Custom node within the material graph.

The resulting (r, θ, φ) are normalized to the $[0, 1]^3$ texture domain and handed to the `SparseVolumeTexture` sampler. We denote the resulting map schematically by $f(UVW_{x,y,z}) = UVW_{r,\theta,\varphi}$ in the material-graph annotations in Sec. A.III in Appendix A, where $UVW \in [0, 1]^3$ is the engine’s three-component normalized texture coordinate vector and the subscript indicates which physical basis the coordinates resolve, with $UVW_{x,y,z}$ referring to the engine’s object space Cartesian frame and $UVW_{r,\theta,\varphi}$ to the simulation’s native radial-poloidal-toroidal frame after the inverse map is applied.

Once the sample has been remapped into the simulation coordinates, the `SparseVolumeTexture` lookup is a direct read from the structured grid on which gKPSP has already stored its data. There is no per-sample search for an unstructured element, and no high-order basis

TABLE 1

Four Omniverse-Side Data Paths Exercised on the gKPSP Electrostatic Potential Demonstration

Path	Resampling	Voxel Grid	Cartesian Loss ^a
(a) scipy VDB	CPU, ~10.6 s/frame	400 × 400 × 160, ~8 to 12 mm	48.9% NRMSE (Table 2)
(b) Warp VDB	GPU, 88.2 ms/frame	600 × 600 × 240, ~5 to 8 mm	41.4% NRMSE (Table 2)
(c) Raw + XAC	Render time	256 × 617 × 193 (native + ghost)	Absent
(d) Kit Warp panel	Render time	256 × 616 × 192 (native)	Absent

^aThe Cartesian Loss column refers to the Cartesian -resampling step. Both render time paths still cast the float64 simulation output to float32 before the GPU upload.

TABLE 2

Field-Space Fidelity of the Lossy Field Representations Relative to the Native Structured Grid ($256 \times 616 \times 192$ gKPSP Electrostatic Potential Snapshot)

Representation	NRMSE ^a (%)	Maximum Error ^b (%)	Fine-Scale Power Retained ^c (%)
Uniform radial resampling	25.1	36.0	74.1
Cartesian $400 \times 400 \times 160$	48.9	56.1	33.8
Cartesian $600 \times 600 \times 240$	41.4	57.5	46.1

^aNRMSE is the error root-mean-square (RMS) as a percentage of the field's own RMS over all native grid points.

^bThe maximum error is the largest pointwise error as a percentage of the field's maximum absolute value.

^cThe Fine-Scale Column is the retained poloidal-spectrum power over the band $m = 41$ to 119.

function to evaluate. This is the key reason why structured grid solver outputs admit a clean engine-side rendering path even though the corresponding unstructured mesh route remains open.

The alternative strategy of pre sampling the toroidal field into a Cartesian bounding box VDB was tested and rejected. Although storage waste can largely be avoided by handling the sparse data with OpenVDB's hierarchical tree, which stores only the active voxels of an otherwise empty bounding box [17], the Cartesian pre-sampling step still allocates the voxel resolution itself on a uniform (x, y, z) lattice that does not follow the torus shape, so the effective voxel pitch on the magnetically interesting region is coarser than the underlying structured grid.

In addition, the Cartesian pre-sampling step degrades the apparent torus boundary because the magnetic-flux-surface boundary no longer aligns with the voxel faces. Performing the coordinate transform inside the material avoids both costs at the price of a small per-sample arithmetic overhead.

Both OpenVDB variants in the preceding paragraphs carry a Cartesian intermediate between the native (r, θ, ϕ) simulation array and the rendered pixel. OpenVDB's sparse VDB representation keeps the per-voxel storage cost of the empty torus bounding box small by collapsing the unused regions into inactive tiles. Those tiles nevertheless incur a tile metadata footprint and per-ray traversal overhead, and the bounding-box framing imposes a coarser than native voxel pitch (see the voxel budget and flux surface misalignment costs noted previously).

What the Omniverse setting additionally exposes is that the toroidal-to-Cartesian resampling itself is irreversible and leaves a residual low-pass footprint at the fine eddy and zonal-flow-streamer scale, with a magnitude that shrinks as the Cartesian pitch tightens but does not vanish.

The coordinate conventions across this path are recorded explicitly. The physics codes producing the

data are right handed, with the simulation frame the right-handed toroidal triad (r, θ, ϕ) . Unreal Engine uses a left-handed Cartesian convention, whereas the Omniverse USD stage is right-handed. These handedness and axis direction differences are absorbed into the coordinate transformation stage of each implemented path. The engine-specific axis mapping is applied together with the inverse map of Eq. (1) in the HSL Custom node on the Unreal side and in the XAC and Warp kernels on the Omniverse side (see Appendix B), and consistency is checked by a cross-path comparison of the same snapshot, requiring consistent placement and orientation of the same field structures across the implemented paths.

This XAC path additionally uses the Y-up convention (torus axis aligned with Y), which differs from the Z-up convention assumed by the material-side remap introduced previously. The choice was local to our Index USD authoring convention for this path rather than a hard USD or Index requirement. A single periodic ghost plane was appended at the high end of each periodic axis, one extra slice on θ and one on ϕ , each replicating the axis-zero plane, so that the trilinear sampler wrapped cleanly across the 2π seam. Without it, the default Index sampler would produce a visible wedge-like seam line when viewed from the side.

The Kit Warp host interface differs from the XAC path in three respects. Geometric constants (R_0 , a , the nonuniform r grid, and $g_{\max}$) are passed as kernel arguments because the Python host can read the ginfold record directly. Periodicity in θ and ϕ is handled by modulo indexing inside the Warp kernel, so no ghost planes are required. Finally, the extension uses the Z-up convention (torus axis aligned with Z) matching the OpenVDB paths and sidestepping the Y-up convention adopted by our XAC implementation in path (c). A systematic audit of every convention pair across all the paths, beyond the implemented ones, remains future work.

The map is implemented independently in the HLSL material node of the Unreal route (see Sec. IV and Sec. A.II in [Appendix A](#)), in the IndeX XAC program, and in the Kit Warp kernel ([Appendix B](#)). The fidelity cost of the Cartesian alternative is quantified in [Table 2](#) (Sec. IV).

IV. RESULTS

This section collects what the implemented paths delivered. It reports the Unreal Engine demonstrations and in-engine observations, the Omniverse four-path fidelity ladder, the quantitative fidelity, overhead, and memory characterizations, and the cross-engine implementation characteristics. The engine-specific implementation and operating details behind these results are documented in [Appendix A](#) and [Appendix B](#).

All the performance readings reported in this section came from a single desktop workstation running Windows 11, Unreal Engine 5.6.1 with Lumen global illumination and ray-marched reflections enabled on the heterogeneous volume material, Omniverse Kit SDK 109.0.1, and a single NVIDIA GeForce RTX 3090 Ti. The same workstation hosted the standalone measurement harness reported later in this section.

Matched scene complexity, lighting, and post-processing settings were used across all the resolution and coordinate variants. Two kinds of readings are quoted. Frame rates are observations of the on-screen frame rate overlay during scene navigation, and therefore are approximate, varying with viewpoint and scene composition. The per-frame GPU times are averages reported by the engine's built-in stat unit profiler at a fixed camera position, recorded to integer-millisecond precision, with an empty scene providing the engine-overhead floor of 4 ms.

The numbers should be read as illustrative of the design choices that follow rather than as a hardware portable benchmark. The frame rate figures characterize the playback budget within which the dynamics of a large-scale scientific computation remain interactively navigable without visible stutter, not a peak hardware capability.

The two kinds of reading just described, the frame rate overlay observations during navigation and the fixed-camera stat unit profiler averages, are the first two of six evidence families on which this section draws. The third is the standalone GPU ray-marching harness, which implements the same sampling mathematics outside either engine and whose measurements are reported later in this section. The fourth is the resampling and preprocessing timing, measured as stated alongside the

Omniverse fidelity ladder that follows. The fifth is the field space fidelity metrics, computed by rebuilding each lossy representation and sampling it back against the native grid ([Table 2](#)). The sixth is the qualitative free-look observation on the Omniverse paths, which carries no per-frame timings.

Comparisons are valid within a family only. The overlay and profiler readings characterize the Unreal scenes on the stated workstation, the harness, which is neither engine, bounds the representation and conversion costs rather than engine performance, and the qualitative Omniverse observations support no quantitative cross-engine comparison.

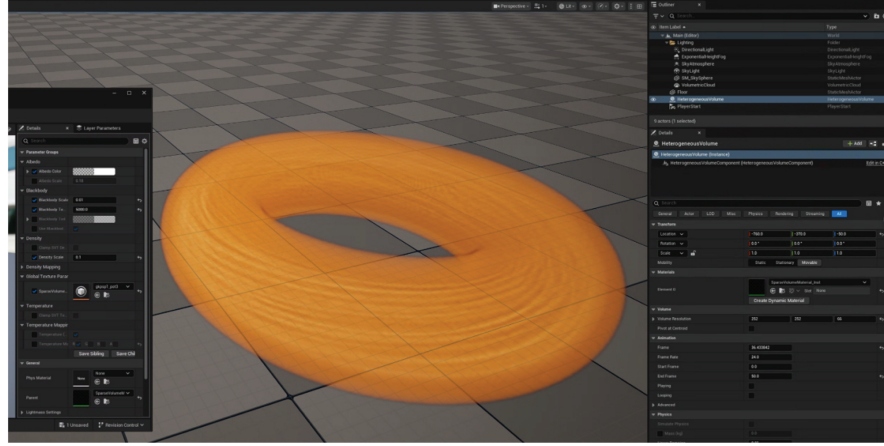
IV.A. Unreal Engine Demonstrations and In-Engine Observations

The first scene in which the upgraded SVT/HV pipeline produced an interactive, device-scale view of a fusion field rendered the whole-tokamak electrostatic potential from a 256^3 Cartesian bounding box stored as an OpenVDB asset. The sparse VDB representation collapsed the empty regions around the torus, and on import, the Unreal SVT exposed the surviving $252 \times 252 \times 66$ active voxels that lay inside the bounding box ([Fig. 3a](#)).

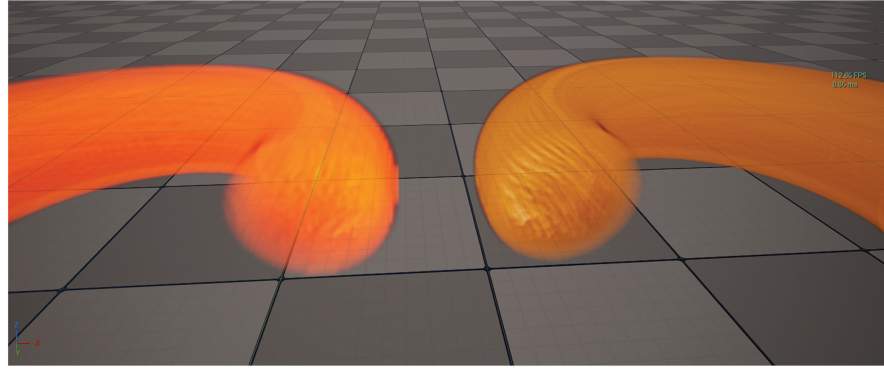
With material settings chosen to expose the ridge-and-trough structure on the torus surface, a loop of 51 successive snapshots was played through the SVT animation block at an internal animation rate of 24 frames per second (fps). This 24 fps figure was a playback parameter of the SVT animation track and was distinct from the engine's own rendering frame rate. During navigation of the demonstrated scene, the frame rate overlay fluctuated at approximately 100 to 110 fps, comfortably above the playback pace.

The 51-snapshot loop was what turned the demonstration into a time-dependent review of the simulated field rather than an isolated still frame. The visible ridge-like structure on the torus surface was consistent with the elongated turbulence eddies that follow the magnetic field lines, a characteristic feature of tokamak plasma turbulence, which the rendering captured despite the relatively coarse voxel budget devoted to the toroidal direction. A quantitative physical attribution of the ridge to a specific parallel mode is left to a separate solver-side analysis on the gKPSP/gKPSP2 simulation output [6,19].

A resolution scan was carried out using the same Cartesian pre-sampled torus at 256^3 and 512^3 voxels ([Fig. 3b](#)). The on-disk OpenVDB files were approximately 6.6 MB per snapshot at 256^3 and approximately 50 MB per snapshot at 512^3 . In separate fixed-camera



(a) Whole-tokamak 256^3 Cartesian OpenVDB asset rendered through the Heterogeneous Volume material. The Details panel on the right reports the $252 \times 252 \times 66$ active voxels surviving sparse compression.



(b) Resolution scan showing the 256^3 (left torus) and 512^3 (right torus) assets side-by-side with a single cut plane through each torus. The 512^3 cut surface resolves the eddy-scale turbulence visibly more clearly than the 256^3 surface. The overlay in this RTX 3090 Ti capture reads 112.86 fps for the two volumes displayed together at the depicted viewpoint.

Fig. 3. OpenVDB/SVT/HV rendering of the gKPSP electrostatic potential at increasing resolution, with (a) the whole tokamak 256^3 asset and (b) the 256^3 versus 512^3 resolution scan with cut planes.

profiler observations, the stat unit GPU time averaged 4 ms for the 256^3 scene and 5 ms for the 512^3 scene against the 4-ms empty scene floor, and during navigation, the frame rate overlay fluctuated at approximately 100 to 110 fps for both scenes.

Taken per scene, these observations did not quantify the cost of displaying both volumes simultaneously. Both tori were shown with a single cut plane exposing the interior of each torus, so that the volumetric structure could be inspected directly rather than only inferred from the surface. The 512^3 cut surface resolved the eddy- and ridge-scale turbulence structure of the gyrokinetic field visibly more clearly than the 256^3 cut surface did, consistent with the expected resolution dependence of the resolved volumetric features and confirming that the OpenVDB/SVT pathway was practical for moderate-resolution fusion field visualization in Unreal Engine 5.6.1 before the more demanding native coordinate run discussed next.

The central positive outcome of this section was the rendering of a gKPSP electrostatic potential field directly on its native curvilinear structured grid without an intermediate Cartesian pre-sampling pass. The simulation's native gKPSP grid was $256 \times 616 \times 192$ voxels in the radial, poloidal, and toroidal directions, respectively. To allow for trilinear interpolation to wrap cleanly across the 2π seams in θ and φ without sampler discontinuities, a single periodic ghost slice was appended along each of the two periodic axes, giving the ghost-augmented $256 \times 617 \times 193$ array that the OpenVDB asset stored, about 30 million voxels per snapshot at an OpenVDB file size of approximately 91.2 MB.

Inside the SparseVolumeMaterial graph, the inverse map of Eq. (1) was applied to each ray sample before the SparseVolumeTexture was read, so the structured grid lookup was reduced to a direct texture fetch. Fig. 4 shows the resulting scene side by side with

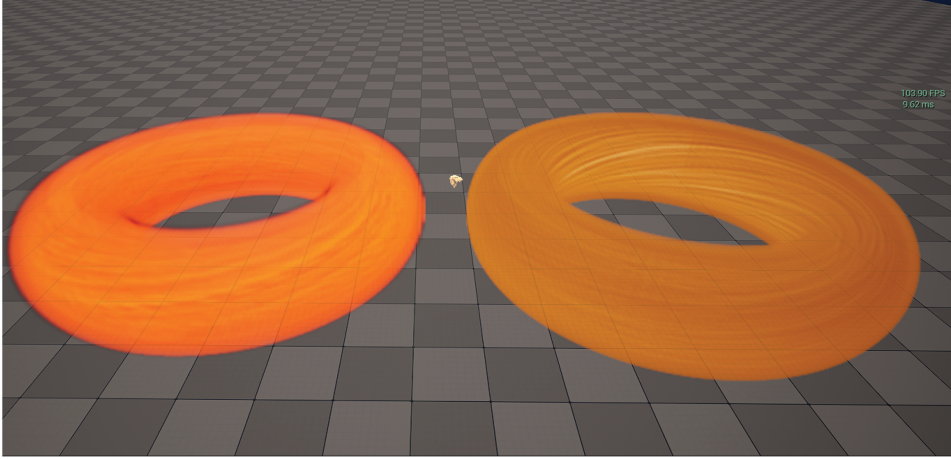


Fig. 4. Central structured grid result showing a side by side rendering of the gKPSP electrostatic potential field, with the 256^3 Cartesian reference on the left (see Fig. 3b) and the native $256 \times 617 \times 193$ curvilinear sampling on the right. The overlay in this RTX 3090 Ti capture reads 103.90 fps at the depicted viewpoint.

the 256^3 Cartesian pre-sampled torus from the previous paragraph. The native coordinate variant exhibits visibly cleaner torus boundaries and a finer field structure, in agreement with the intuition that the structured grid is densely populated where the device geometry actually carries data, whereas the Cartesian box wastes voxels on empty regions surrounding the magnetic flux surfaces.

In the resulting scene, the frame rate overlay fluctuated at approximately 100 to 110 fps during navigation, comparable to the Cartesian pre-sampled scenes discussed previously, and the fixed-camera stat unit GPU time averaged 4 ms, equal at the integer-millisecond precision of the recorded averages to the empty scene floor and to the 256^3 Cartesian scene. Because the scenes were not controlled counterparts of one another and the averages were rounded to whole milliseconds, these observations did not isolate or bound the coordinate conversion overhead, for which the controlled characterization was the harness measurement

reported later in this section, but no separable conversion penalty appeared under this engine configuration.

These readings indicated that the pipeline can deliver high-fidelity simulation results to the user as an interactive scene without a perceptible rendering delay. The underlying geometric and performance result, a toroidal native field of the full ghost-augmented size stated previously rendering interactively under the inverse coordinate map without Cartesian pre sampling, was the structured grid demonstration that motivated this work.

The SVT/HV route exposed two complementary interactive controls that turn the structured grid scene in Fig. 4 into an interactively inspectable one. The first is a threshold window on the rendered value range, and the second is a movable clip box that the user drives directly in the editor while the field is being rendered. The threshold window, narrowed in the demonstration to $val_{min} = 0.22$ and $val_{max} = 0.23$ (Fig. 5), gives the user an approximate 3D

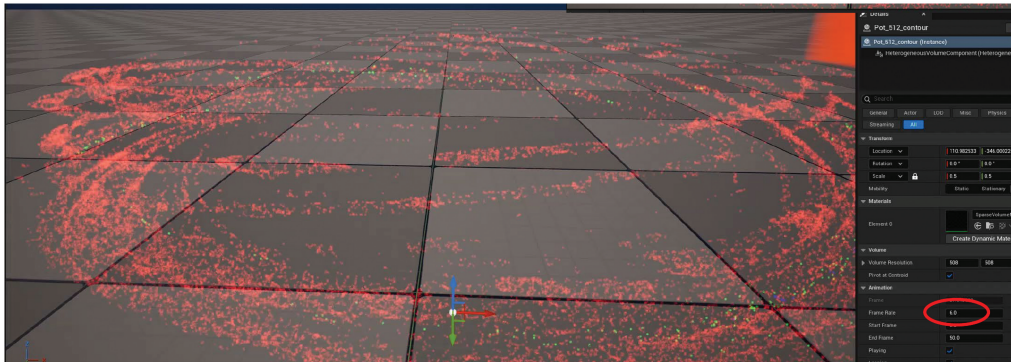


Fig. 5. Threshold window filtering scene with the Details panel that the operation drives. The narrow admissible band $val_{min} = 0.22$, $val_{max} = 0.23$ (highlighted with the red ellipse on the right-side Details panel) collapses the rendered volume into thin specks that we observed to be organized into laterally coherent arc-like bands once playback was slowed for inspection.

contour-like view of the field that can be tuned in real time, and the clip box lets the user slide an arbitrary cross section through the field, making it easy to inspect data on a chosen cut plane during analysis.

Because the SparseVolumeMaterial pin set in Unreal Engine 5.6.1 does not expose an opacity input, the clip box masking is routed through the emissive color and extinction outputs instead of an opacity mask. The control wiring and the operating procedure for both controls are documented in Sec. A.IV in [Appendix A](#).

IV.B. Omniverse Demonstrations: The Four-Path Fidelity Ladder

The Omniverse pathway documented in this section targets the visualization layer of the Korean WILL/V-KSTAR fusion digital twin platform [3,4]. The present demonstration exercised that pathway on the same KSTAR equilibrium-derived mesh and gKPSP electrostatic potential snapshot sequence used in the Unreal demonstrations earlier in this section, with the four paths described in the following progressing from an Omniverse Cartesian OpenVDB representation generated from the same sequence to two render time native grid evaluations.

Co-registration of these volumes with the WILL/V-KSTAR scene graph alongside the KSTAR engineering CAD and operator room infrastructure is identified as the next integration step rather than as completed work in the present study. The Omniverse-side stages reported in the following, therefore, are standalone scenes prepared for that subsequent integration.

The Unreal Engine demonstrations earlier in this section established that the proposed mesh-to-volume data path supports interactive volume rendering, time-dependent playback, and material-driven coordinate handling on a single rendering substrate. This section reports a second front end built on NVIDIA Omniverse, in which the same simulation output reaches an OpenUSD-based digital engineering scene through four distinct data paths that together form a fidelity ladder ([Table 1](#)).

Path (a) registers a Cartesian OpenVDB representation of the gKPSP snapshot sequence at the resolution shown in [Table 1](#), prepared through the same conversion principles as in Sec. III, but as an independently produced Omniverse-side asset rather than a shared on-disk file. Path (b) replaces the CPU-side `scipy.RegularGridInterpolator` resampling with a NVIDIA Warp [30] GPU kernel, substantially faster at matched grids (quantified in the following), makes the denser Cartesian voxel lattice shown in [Table 1](#) a practical operating point.

The backend and the output Cartesian resolution were independent choices. Path (c) bypasses the Cartesian intermediate entirely and feeds the (r, θ, ϕ) grid to the IndeX backend [31] in angularly native form, with the slightly nonuniform radial axis linearly resampled onto a uniform axis at conversion time, evaluating the inverse toroidal transform per ray-march sample inside an IndeX XAC program [32].

Path (d) implements the same render time inverse toroidal transform inside a custom Omniverse Kit extension, where a NVIDIA Warp kernel ray marches the native (r, θ, ϕ) array on the GPU each frame and writes the result into a Kit panel through `omni.ui.ByteImageProvider`. Beyond the colormap and threshold controls that the IndeX graphical use interface (GUI) already provides in path (c) without XAC recompilation, this route adds custom playback, slice, camera, and kernel parameter controls authored directly in Python.

Paths (a), (b), and (c) are authored as USD stages that share the same scene graph template and differ only in the Volume prim's field binding, so an analyst can switch operating points by opening a different stage without repreparing the underlying data. The decision to present multiple paths within a single section was deliberate. Rather than committing to one Omniverse implementation, this work documents an engine-side data path family and reports the engineering trade-offs that justify each step on the ladder. Throughout this section, Omniverse is treated as a complementary digital twin substrate rather than as a substitute renderer. Quantitative inter-engine performance comparisons remain deferred to Sec. V.

The CPU-side path (a) resamples each snapshot onto a $400 \times 400 \times 160$ Cartesian grid (about 2.5×10^7 voxels at an approximately 8- to 12-mm anisotropic pitch, ~ 12 mm along the horizontal x - and y -axes that span $2(R_0 + a)$ and ~ 8 mm along the vertical z -axis that spans $2a$) with the single-threaded `scipy.RegularGridInterpolator` from SciPy 1.17.1, at roughly 10.6 s per frame on the same RTX 3090 Ti workstation.

The GPU-side path (b) performs the same toroidal-to-Cartesian resampling through a NVIDIA Warp kernel and raises the grid to $600 \times 600 \times 240$ ($\sim 8.6 \times 10^7$ voxels at an approximately 5- to 8-mm pitch), taking 17.2 ms of kernel time plus 71.0 ms of device-to-host transfer per frame on the same machine. The transfer cost was dominated by pageable memory copies on this workstation, while the kernel alone sustained $\sim 5 \times 10^9$ samples per second, so that at matched grids, the full GPU path ran about 400× faster than the CPU baseline.

This kernel-side speed is what made the denser grid a practical operating point for batch conversion. This was a choice of operating point, not a property of the Warp backend itself. The Warp kernel output was cross checked against the scipy reference to $\sim 10^{-6}$ in absolute residual at $400 \times 400 \times 160$. Because both variants followed the same FloatGrid schema, they shared the same Omniverse tooling (timeline scrubbing, transfer function authoring, and threshold filtering) without modification, and the Warp variant tightened the voxel pitch by a factor of $\sim 1.5\times$ per axis without reaching the native poloidal resolution (~ 3.2 mm near the mid radius).

The residual gap motivated the render time native grid paths introduced later in this section, and these two OpenVDB variants also served as the baseline against which those paths were compared (see Fig. B.1).

The Omniverse-side demonstration was built around NVIDIA's Index volumetric backend, exposed through Omniverse's RTX-Scientific renderer [31,33] because Index targets scientific volume data and provides built-in transfer function authoring, arbitrary cross-section planes, ambient occlusion, and multi-GPU domain decomposition without per-scene reimplementations. The two interactive Unreal-side controls introduced earlier in this section carried over to Omniverse native operations directly.

The narrow admissible window in Sec. IV.A (Fig. 5) is expressed in Index through the opacity ramp of the transfer function, so that voxels outside the window contribute neither emission nor extinction (Fig. B.2a), and Index exposes arbitrary cross-section planes as first-class scene primitives, which subsume the BoxMask3D dissolution behavior in Sec. A.IV in Appendix A without the SVT-Opacity workaround required by Unreal Engine 5.6.1, so that the analyst can move axis-aligned cross-section planes through the torus and inspect the interior field without modifying the VDB sequence (Fig. B.2b).

Index additionally exposes a transfer function editor that maps the normalized magnitude $|\phi|/\max|\phi|$ stored in the VDB asset to color and opacity through a one-sided ramp (e.g., a black-to-orange amplitude scale). Because this magnitude encoding collapses sign information at VDB write time, recovering the explicit positive/negative split that the Unreal route in Sec. III.B achieves on the signed field was deferred to the render time native grid paths described in the following, which retain the natively signed scalar and can be colored with a diverging map whose alpha minimum sits at the zero amplitude region.

Ambient occlusion and Index's built-in shadowing improve depth perception of strong-amplitude eddies along the toroidal direction, so the apparent ridge structures noted earlier in this section remain visible under Omniverse rendering and strengthen their interpretation as a feature of the data rather than of one material graph. In our runs, the Index backend sustained interactive frame rates suitable for free-look exploration across both OpenVDB variants discussed in the previous paragraph. Quantitative per-frame timings depend on the Index sampling distance, multi-GPU configuration, and ambient-occlusion settings, all of which were left at their Composer-side defaults, and a controlled, matched hardware performance study, therefore, is positioned as future work in Sec. V.

The render time paths (c) and (d) in Table 1 remove this intermediate by keeping the simulation array in its (r, θ, ϕ) form and evaluating the inverse map of Eq. (1) at every ray-march sample, returning a background value where $r > a$. The array is angularly native in both paths, with path (c) carrying the radial axis pre-resampled to a uniform grid at conversion time and path (d) carrying the original nonuniform radial grid with periodicity handled by in-kernel modulo indexing. The rendered image then inherits the full angular resolution of the simulation in both paths and additionally preserves the native nonuniform radial spacing in path (d), rather than that of the intermediate Cartesian VDB.

The interaction between Index's arbitrary cross-section planes and the XAC sampling stage was less clean in our demonstration. The slab boundary visually showed as distributed noise rather than a definite cut surface, and a faithful slice presentation in the XAC path required either kernel-side slab gating or a separate validation pass, so the cleaner Cartesian-VDB behavior of paths (a) and (b) was retained when an explicit cross section was required.

Across all four paths, the analyst can choose an operating point between fidelity and operational convenience without re-preparing the original simulation data, working from the same standalone OpenUSD stage in paths (a), (b), and (c) or alongside it in path (d). The broader positioning of these paths against the Unreal front end demonstrated earlier in this section, and the outlook on a hardware matched comparison between the fidelity ladder rungs, are taken up in Sec. V.

IV.C. Quantitative Fidelity, Overhead, and Memory

This revision added a quantitative characterization of the native coordinate representation against its Cartesian

resampled baselines. The measurements were carried out with a standalone GPU ray-marching harness that implemented the same sampling mathematics as the in-engine paths, with identical camera and transfer function, and a fixed step count with early ray termination disabled, so that the comparison isolated the representation and conversion costs from engine-specific optimizations.

The harness figures, therefore, were conservative upper bounds relative to the in-engine readings reported earlier in this section, which benefited from hardware texture filtering and early ray termination. All the tabulated harness figures were obtained on the same single-GPU workstation (a NVIDIA GeForce RTX 3090 Ti) as the in-engine observations discussed previously, under the separate, controlled measurement conditions stated earlier.

Fidelity was evaluated in the field space, taking as the reference the native grid, which was the simulation output itself and what the native coordinate paths sample directly. Each lossy representation was rebuilt exactly as the pipeline built it and sampled back at all the native grid points (Table 2). The errors were expected rather than anomalous. A 400-class Cartesian voxel is 12.2 mm across, while the field's own spectra carried power down to half-wavelengths of ~ 2.4 mm radially (the native grid's resolution limit) and ~ 4.3 mm poloidally, a three- to fivefold undersampling of the turbulence.

The loss was concentrated in the fine scales. Over the poloidal band $m=41$ to 119, which carried the native field's 90th to 99.99th percentile of cumulative poloidal power, the retained power dropped to the values listed in Table 2, while the large-scale field survived (the 400-class reconstruction still correlated at $r=0.88$ with the original). These figures characterized what the renderer sampled. The photometric accuracy of the rendered image was

governed by visualization choices, namely the transfer function range and shading conditions, such as lighting and surface material, and was outside the scope of this paper (see also the interpretation caveat in Sec. V).

The render time cost of the native coordinate path was one coordinate conversion per ray sample, executed before the field value was fetched (Table 3). Relative to an identity map control that read the same array with the conversion removed, the conversion cost roughly 60% to 65% of the control frame time at 1280×720 , with the absolute overheads at both tested resolutions listed in Table 3, and the native path remained interactive at both resolutions at the frame rates given in the table.

The identity map control itself did not grow between the two tested resolutions (Table 3). The fixed-step ray march was a latency bound kernel that did not saturate the workstation GPU at 1280×720 , so the additional 1920×1080 pixels were absorbed by otherwise idle capacity, whereas on the desktop-class GPU of the supplementary checks, the control scaled approximately in proportion to the pixel count. The tabulated overheads, therefore, were differences against a per-resolution measured control, and per-GPU overheads should be compared against their own controls rather than across GPUs.

The Cartesian route, by contrast, paid a separate per-snapshot resampling pass before rendering, with the kernel and device-to-host transfer times again listed in Table 3, in addition to the fidelity loss in Table 2. On this GPU, the comparison inverted at the denser grid, where a 600-class Cartesian volume rendered slower than the native path itself (46 ms versus 29 ms per frame at 1280×720). In supplementary checks, the same harness ran unchanged on a second, desktop-class NVIDIA GPU, which reproduced the native path frame rates and resampling kernel times within about 12%.

TABLE 3

Render Time Cost of the Native Coordinate Path on the Standalone GPU Ray-Marching Harness (Fixed Step Count, Early Ray Termination Disabled, Single NVIDIA GeForce RTX 3090 Ti Workstation)*

Quantity	1280×720	1920×1080
Native path frame time	29 ms (34 fps)	48 ms (21 fps)
Identity map control frame time	18.0 ms	16.4 ms
Per-sample conversion overhead	+11 ms	+31 ms
Cartesian 600-class frame time	46 ms	51 ms
Cartesian 400-class resampling (per snapshot)	4.6 ms kernel + 21 ms transfer	

*The conversion overhead was isolated by an identity map control that read the same array with the coordinate conversion removed. The Cartesian route additionally paid a per-snapshot resampling pass before rendering.

This limited two-GPU comparison did not establish portability across vendors or broader architectures. The workload was dominated by per-sample-dependent memory reads (coordinate conversion, then a gather) rather than by architecture-specific units, with no ray tracing or tensor cores used, so only weak sensitivity to GPU architecture was expected in this regime, consistent with the supplementary checks.

With regard to memory, the native and Cartesian representations were not an apples-to-apples comparison, since they stored the field at different resolutions for different purposes. The Cartesian representation was sampled at a user-chosen resolution and stored in the sparse OpenVDB format, which omitted the empty space around the torus, so it was small (the per-snapshot file sizes are recorded in Sec. IV.A).

The native representation kept the whole plasma domain at the simulation’s own resolution, so it was larger at ≈ 121 MB, resident as a dense single-precision array stored on disk as the ghost-augmented OpenVDB asset, whose file size is likewise recorded in Sec. IV.A. For scale, a uniform Cartesian grid matching the native radial spacing would need 8.6 GB at the mean spacing (2.43 mm) and 25 GB at the finest interior spacing (1.70 mm), i.e., $71\times$ to $206\times$ the native resident footprint. An equal-fidelity Cartesian equivalent, therefore, would not be practical, which is the substantive content of the comparison.

IV.D. Cross-Engine Implementation Characteristics

Finally, Table 4 collects the characteristics of the two engine-side implementations that proved noteworthy during this work. The table records the observed implementation characteristics rather than an evaluation of the two commercial engines, which is outside both the scope and the authority of this paper.

V. DISCUSSION AND OUTLOOK

The work reported in the preceding sections established a fusion-specific visualization layer that takes mesh and field data produced by gyrokinetic solvers and renders them inside two distinct real-time engines. Three concrete elements carried the demonstration. The first was the in-engine PUMI .smb loader in Sec. III.A, which provided an existence proof for solver-side mesh ingestion without a MPI dependency. The second was the SVT and HV route, discussed in Sec. III.B, which removed the signed field ray accumulation, run time PNG playback, and internal inspection limitations of the early PVT path documented in Sec. A.II in Appendix A.

The third was the four-path Omniverse fidelity ladder in Sec. IV.B, which climbs from Cartesian OpenVDB baselines produced by the same conversion principles (as independently produced Omniverse-side assets) through a Warp-accelerated Cartesian variant into two render time native grid paths fed by a raw structured container, the IndeX XAC program and the in-Kit Warp panel, attached to a standalone OpenUSD stage prepared for eventual co-registration with the WILL/V-KSTAR platform.

The central scientific result was the structured grid native coordinate rendering in which a $256 \times 617 \times 193$ gKPSP electrostatic potential volume was rendered directly in (r, θ, ϕ) at interactive frame rates (Sec. IV.A), with the inverse engine-XYZ-to-toroidal map resolved per ray sample inside the HV material in Unreal Engine and inside an IndeX XAC program in Omniverse. Threshold window and clip box controls turned the rendered volume into an interactively inspectable scene from which a user can extract approximate 3D contour-like views.

The components used here are individually well established. PUMI serves unstructured mesh handling, OpenVDB sparse volume representation, Unreal Engine sparse volume rendering, and Omniverse/OpenUSD digital

TABLE 4

Noteworthy Characteristics of the Two Engine-Side Implementations Observed During This Work, Recorded as Implementation Characteristics Rather Than as an Evaluation of the Engines

	Unreal Engine (Appendix A)	Omniverse (Appendix B)
Volume rendering backend	SVT + HV	RTX + IndeX XAC + Kit Warp
Data path	OpenVDB $\rightarrow$ SVT (+ PVT flipbook)	OpenVDB baselines + raw container
Native render time transform	HLSL Custom node, Eq. (1)	XAC kernel + Kit panel
Interactive inspection	Threshold window + clip box	Kit panel (playback, camera, slices)
Extension mechanism	C++ loader + Blueprint + HLSL	Python Kit extension + XAC CUDA
Performance readings	Sec. IV.A (per scene)	Qualitative (interactive free look)

engineering scene assembly, with the use of Omniverse/OpenUSD for fusion CAD scene composition itself exemplified by the UKAEA STEP digital twin work [12].

Their joint use as a fusion-specific data path was the contribution. Three observations from the Omniverse front end in Sec. IV.B unpacked this joint role. First, the OpenVDB-plus-OpenUSD pairing acts as an engine-neutral intermediate representation between the gyrokinetic solver output and the digital engineering scene composition, so that both engine front ends are fed from the same field representation without per-engine changes on the solver side (the engine-side assets themselves are produced in parallel rather than shared on disk, see Sec. IV.B).

Second, the Warp-accelerated Cartesian variant of path (b) showed that the presampling cost can be moved from CPU to GPU without modifying that interface. Third, the render time native grid paths (c) and (d) showed how the inverse toroidal map can be carried either inside a scientific renderer program (the Index XAC kernel) or inside a host-side GPU kernel (the in-Kit Warp panel), in both cases leaving the OpenVDB-plus-OpenUSD interface untouched as the baseline.

Together with the material-side ray-marching path in Sec. III.C, this gives a substrate-independent picture of where the coordinate handling can live without disrupting the upstream data path. The same field representation, therefore, can be used both for the renderer-facing volume inspection, and in a follow-up integration, for collocation with KSTAR-scale CAD geometry inside a USD scene graph.

Two further points complement this positioning. First, the structured grid result was the part of the demonstration that exploited the geometry of the specific solver class, gKPSP, rather than a generic Cartesian volume. The voxel budget saving and the closer correspondence between the original field samples and the rendered samples both followed from keeping the data in (r, θ, ϕ) (see Sec. III.C), and together they justified treating the material-side coordinate transform as an architectural choice rather than a convenience.

Second, the proposed workflow should be read as a digital twin-enabling visualization layer rather than as a digital twin in the closed-loop sense increasingly used in the fusion deuterium-tritium literature [1,2]. The demonstrated path was one way. Mesh files in the PUMI format and field snapshots in OpenVDB were consumed by the engine, and the resulting scene was a viewer rather than a controller, though the viewer supports interactive data probing through the threshold window and clip box controls discussed in Sec. IV.A.

As stated in Sec. I, we reserve the term digital twin for systems that include automated feedback to the simulator and live coupling to experimental operation data. The present contribution supplies the visualization and scene integration tier on which a fuller digital twin assembly could later be built.

The scientific interpretation of the volume-rendered fields remains a real limitation. Transfer function choices, opacity mappings, extinction coefficients, ray-accumulation strategies, and threshold values can all noticeably change which structures appear visible in the rendered image. The threshold filtered contour-like view, in particular, revealed coherent banded patterns whose physical interpretation has not yet been validated against simulation diagnostics, such as turbulent eddies, zonal flows, or wavenumber spectrum analyses.

The same caveat applies to the apparent ridge-like structure visible on the whole-tokamak rendering in Fig. 3, whose attribution to a specific parallel mode also requires a separate solver-side $k_{\parallel}$ analysis. For this reason the rendered images, including the whole-tokamak ridges, the threshold view, and the clipped scene, should be treated as exploratory visualizations rather than as quantitative output until they are calibrated against solver-side analysis or against independent diagnostics.

This caveat is not specific to the present implementation. It follows from the way classical direct volume rendering combines emission and absorption along view rays [34,35] and from the practical guidance for large-volume rendering in game engines summarized by Schlüter et al. [8]. A natural direction for closing this gap is to develop in-engine physics analysis tools that operate on the same OpenVDB grid as the renderer, compute scalar diagnostics in real time, and surface their output as side-panel widgets next to the rendered volume. In this way the interactive volume scene in Sec. IV.A could be progressively upgraded from an exploratory viewer to a quantitatively grounded analysis environment inside Unreal Engine and Omniverse alike.

The mesh-and-field ecosystem inside which the present visualization workflow sits is broader than the gyrokinetic case used here, and the codes that compose it span a wide range of mesh and field configurations. The gKPSP family alone, whose three solver generations and their meshes are introduced in Sec. II, illustrates the breadth on the mesh side. The visualization challenge grows from a structured grid field read (gKPSP) to high-order discontinuous Galerkin field rendering on a 2D

axisymmetric poloidal plane mesh (gKPSP2), and finally, to a 3D field-aligned volume of curved high-order elements that sit above the linear rasterizer assumptions of the engine pipeline (gKPSP3).

Among the other Korean fusion codes that share the PUMI substrate, ERRHAI solves a 3D eddy current problem on a true 3D volume mesh whose node-centered field is the magnetic-vector-potential solution [36]. NuBDeC tallies neutral beam fast-ion losses on a 2D plasma-facing surface mesh, using that mesh as a tally support rather than as a 3D field carrying volume, and relies on a spatial hash collision detection structure rather than on volumetric ray accumulation [26,37].

POCA performs particle orbit calculations whose mesh role is the diagnostic line and perturbation field context [38]. The V-KSTAR/WILL platform organizes the same PUMI-managed mesh assets into a digital twin scene graph [3,4]. Finally, the AutoFRD reactor design pipeline operates on CAD-derived engineering mesh geometry rather than on physics fields.

This substrate rests on common mesh tooling infrastructure comprising the FMDB lineage parallel mesh database that ultimately became PUMI [5,39], the Gmsh generator used for engineering domain meshing [40], and the Simmetrix SimModeler suite for geometry-based mesh modeling [41], with CAD ingestion via CadQuery and the Open CASCADE Technology kernel [42,43].

The data path delivered by the present workflow plugs into a stack that already has provenance across the Korean fusion digital twin program. The visualization workflow described here is one component of that ecosystem, providing the interactive scene endpoint for any of the PUMI-based meshes equipped with appropriately formatted field snapshots. The other applications were deliberately treated as motivation and as future integration targets rather than as full case studies, because demonstrating each one would require its own validation and would dilute the central mesh-to-volume contribution.

Two forward-looking observations follow from this enumeration. From a mesh perspective, faithfully representing the curved surfaces and edges of a 3D field-aligned volume, the case of gKPSP3 in particular will require importing high-order element shape functions into the engine, rather than refining a linear element approximation until the curvature is hidden. From a field perspective, p -refinement carried by high-order basis functions on top of those elements is a natural complement, and the joint (h - and p -) refinement story is one of the tractable next steps for moving the unstructured mesh branch of the workflow off the deferred list in Sec. II.

Several engineering limitations remain on the path between the demonstrated workflow and a more complete fusion data infrastructure. The PUMI-side loader currently ingests only mesh geometry, with node- or element-centered field attachments and high-order Discontinuous Galerkin (DG) basis evaluations both deferred (Secs. II and III.A). Lifting the latter would require a custom HLSL pixel shader basis evaluator.

Partitioned mesh ingestion has not yet been implemented on the engine side, which was acceptable for the demonstrated single-part workflow because the global gyrokinetic mesh was generated as one .smb container before solver-side partitioning and the PUMI utilities could remerge a partitioned mesh on demand [5], but truly multidomain workflows will eventually require multipart support.

The natural successor solver gKPSP3, with its curved high-order elements and nonconforming inter-wedge interfaces, will add a curved element rendering challenge inside the engine's rasterizer pipeline [20]. Coordinate handedness, where the Unreal convention is left-handed while the simulation conventions are right-handed toroidal, is handled explicitly in the coordinate transformation stage of each implemented path and checked by a cross-path comparison of the same snapshot (Sec. III.C). A systematic audit of every convention pair beyond the implemented paths remains future work.

A controlled hardware-matched comparison between the fidelity ladder rungs in Sec. IV.B, in particular the IndeX XAC path, the in-Kit Warp panel, and the Cartesian-VDB Unreal path in Sec. IV.A, is a related validation gap. The in-engine Omniverse observations were made in a separate environment and without controlled per-frame instrumentation, so a matched settings benchmark on a single common hardware configuration, with controlled IndeX sampling distance and an ambient occlusion configuration, remains to be carried out.

The standalone harness study reported in Sec. IV.C isolated the per-sample conversion cost and its hardware sensitivity under controlled conditions, but it did not replace this in-engine matched settings benchmark, which involves engine-specific sampling and shading settings. Larger-scale time series management, beyond the 51-snapshot animations used in the present demonstration, will require dedicated streaming and caching strategies.

Finally, a GPU-side mesh search for unstructured field sampling, that is, locating for a given view ray sample the unstructured element that contains it, is the natural next step for moving the high-order DG branch off the deferred list. This is consistent with the broader

need for GPU mesh applications and for mesh-based data transfer between codes within a prescribed error tolerance, as articulated in the parallel coupler for multimodel simulation development [44], and could plausibly draw on the three GPU point location families already itemized in Sec. II, together with the cluster-based culling and software rasterizer techniques developed for the engine's virtualized geometry system [22].

Looking further ahead, the same scene content is compatible in principle with multinode and multiprojector deployments through the engine's nDisplay and Multi-Process infrastructure [9,45], opening a path toward immersive digital twin rooms of the kind already discussed in the fusion visualization literature [2,10,11]. Complementary GPU volume back ends, such as NVIDIA IndeX [31], provide further options for the out-of-core handling of larger field data sets.

Real-time streaming of the solver output, as opposed to the precomputed snapshot playback demonstrated here, is not a meaningful target for this data source. A high-fidelity gyrokinetic run advances at seconds to tens of seconds per time step and takes weeks to a month overall, whereas the visualization already draws more than 10 fps, so the pipeline deliberately targets the post-hoc interactive exploration of precomputed snapshots.

With regard to scaling, larger gyrokinetic data sets require supercomputer resources and long run times, so no empirical scaling claim is made here. The present data set did, however, resolve the ion-gyro-radius scale with a radial grid of 256 points ($256 \times 616 \times 192$ overall), representative of the upper range of published gyrokinetic simulations, and since the method only added a per-sample coordinate conversion to the standard volume rendering, no structural obstacle is expected at higher resolutions (compare the multigigavoxel SVT demonstration surveyed in Ref. [8]).

Because each component of a vector field can be represented as a scalar field, the demonstrated scalar path extends componentwise to vector data (see Sec. I). Dedicated vector representations that convey magnitude and direction simultaneously, such as glyph- or streamline-based techniques, are future work, with the magnetic vector-potential fields of Eddy cuRREnt Analysis on Hierarchical Inductance (ERRHAI) [46] noted previously as a concrete target. More broadly, the methodology is portable to any visualization platform that provides a programmable per-sample shading hook together with a volume asset path.

The inverse map of Eq. (1) was implemented independently in three programmable sampler environments in

this work alone, namely a HLSL Custom node in Unreal Engine, a CUDA XAC program in NVIDIA IndeX, and a Python/Warp kernel in the Omniverse Kit extension. The structured grid results in Sec. IV.A suggests that the most immediate scientific extension of this work is to broaden the class of structured grid solvers whose native coordinates can be hosted by the material-side inverse map, while pursuing the high-order DG branch on the unstructured side as a separate research thread.

Despite the limitations enumerated previously, the demonstrated workflow provides a practical bridge between high-fidelity fusion simulations and interactive digital twin environments. By preserving the connections among the mesh geometry, field data, native coordinate representation, and visualization control, the workflow can make complex 3D fusion simulation results more accessible for analysis, communication, and future integrated digital twin development. We view the present demonstration as a stepping stone in that direction, a documented and reproducible visualization layer on which fuller fusion digital twin assemblies can subsequently be built.

APPENDIX A

UNREAL ENGINE IMPLEMENTATION AND OPERATION

This appendix documents the Unreal Engine-side implementation details and operating procedures that support the approach in Sec. III and the demonstrations in Sec. IV. All the Unreal-side operations described here were carried out in the environment stated in Sec. IV, a single desktop workstation running Windows 11 and Unreal Engine 5.6.1, with Lumen global illumination enabled on the heterogeneous volume material and a single NVIDIA GeForce RTX 3090 Ti.

A.1. PUMI .smb LOADER INTERNALS

Concretely, a C++ loader for PUMI native .smb files runs inside Unreal Engine, exposed through a custom Details Panel customization. The user points a PumiActor, an Unreal Engine actor that carries the loaded mesh in the scene, at an .smb file on the disk, and the loader parses the binary container in process. In the small parametric field on mesh test noted in Sec. III, a single tetrahedral element was assigned arbitrary per-vertex scalar values and rendered with an in-material HLSL linear interpolation routine that sampled the resulting texture inside the material shader.

A.II. PVT FLIPBOOK PATH AND SIGNED FIELD ENCODING

For the field pathway, the initial implementation followed a flipbook-plus PVT route, developed starting from Brucks’ shader-based volumetric ray marcher [23]. A Python flipbook generator was written to convert gKPSP electrostatic potential snapshots into PVT-compatible texture atlases, in which 16×16 uniform sampling planes rendered at 300 dpi were tiled into a single PVT texture page. The resulting atlases were sampled by a custom Unreal material graph that performed camera vector-driven parallax occlusion-style ray marching, and a Blueprint For Loop with Delay cycles, a 51-frame sequence at 0.05 s per step.

This implementation confirmed that time-dependent fusion field data could be visualized inside an engine scene, but it also exposed three limitations that motivated the later upgrade. First, the PVT atlas layout reduced the sampling between the adjacent flipbook slices to a manually blended linear interpolation (linear inter-slice blurring), which mismatched the higher-order discontinuous Galerkin basis function expansion of the underlying gKPSP2 simulation [6].

This particular interpolation behavior was a property of the flipbook-atlas sampling pattern adopted in our material graph rather than of the PVT format per se, since a custom ray marcher could, in principle, implement higher-order intra-cell sampling at the pixel shader level. Reaching parity with gKPSP2’s high-order DG basis would, however, require either a custom pixel shader evaluator that reconstructs each element’s basis functions on the fly, or a sufficiently refined background mesh on which the engine’s standard linear (barycentric) rasterizer interpolation becomes a faithful approximation

(see Sec. II for the rasterizer-side argument). Neither was undertaken in this work.

Second, because each time step of the time-evolving field had to be streamed into the engine at run time, snapshot ingestion was performed through Import File as Texture 2D on every step of the playback sequence. This run-time PNG import path introduced a perceptible playback delay between the snapshots. Third, ray accumulation in the grayscale space caused the encoding of $\phi = 0$ to add along the ray path, blurring the boundary between the positive and negative regions of a signed scalar field. Together these issues indicated that the shader graph PVT path was a workable demonstration substrate, but not a satisfactory representation for sparse, signed, 3D fusion fields.

The treatment of signed scalar fields is a central issue for scientific visualization in this engine context. The grayscale PVT path documented previously suffered a mid-tone wash that blurred the boundary between the negative, zero, and positive domains. Because the value 0.5 that encoded $\phi = 0$ was accumulated along the ray for every near-zero sample, the rendered volume could not distinguish the vacuum from the zero crossing of a signed field [23].

We note that the pathology is a property of our particular grayscale accumulation encoding rather than of the PVT format as such. To overcome it within the PVT framework, the encoding was reformulated into an RGB-based PVT in which the red and blue channels, respectively, encoded the value $+1$ (mapped to $+\phi_{\max}$) and the value -1 (mapped to $-\phi_{\max}$), while the per-voxel opacity was taken as the average of the RGB channels.

With this color-coded variant, the sign of the scalar is carried by hue contrast rather than a brightness gradient, and the mid-tone wash of the grayscale variant

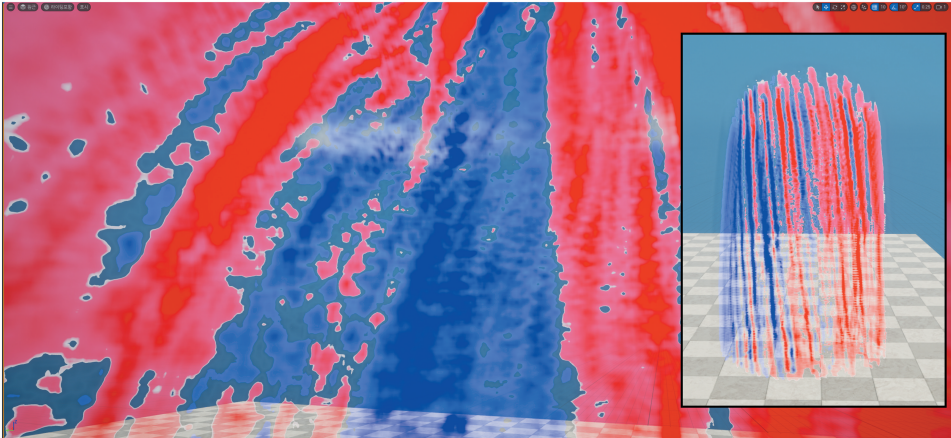


Fig. A.1. Picture-in-picture rendering of the RGB-PVT pathway on a $1/8$ toroidal wedge. The top-right inset (framed) shows the wedge from the outside, while the main panel shows the same wedge from inside the volume actor.

disappears. The resulting scene (Fig. A.1) was the RGB-PVT solution adopted for the flipbook pathway. The OpenVDB-based implementation then removed the ambiguity at a deeper level, at the material level itself.

Density, extinction, and emission scales, transfer function mapping, and per-voxel value window thresholding were exposed as independent parameters of the SparseVolumeMaterial instance through Dynamic Material Instances and Material Parameter Collections (MPCs), following the SVT/HV pattern surveyed for large-volume rendering in Unreal Engine 5 [8]. The same material instance was reused across all the resolution and coordinate variants reported in Sec. IV.A.

A.III. SVT AND HV SETUP

In Unreal Engine 5.6.1, OpenVDB files produced by the workflow in Sec. III.B were imported as SVTs and bound to HV materials so that volumetric ray sampling and shading were handled by the engine rather than by a custom flipbook material graph [15,16]. The starting point of the visualization, therefore, was the engine-shipped HV material template that we refer to throughout this section as the SparseVolumeMaterial (see Fig. A.2). A single template was duplicated and exposed through a Material Instance, and the resulting parameter group was used to set the density scale, extinction, an emission-like blackbody term, and field-dependent visual mappings.

The underlying gKPSP electrostatic potential data were produced in 64-bit precision and written to OpenVDB as 32-bit floating-point values. When the resulting OpenVDB file was imported into Unreal, both Attributes A and

Attributes B channels were configured as 32-bit float and read in at that precision, so that no quantization was introduced beyond the solver-to-OpenVDB step itself [17].

The practical outcome was that all the visualization controls discussed in the remainder of this section acted on a single material instance and did not require recompilation or re-import of the underlying field. A simple complementary control on the engine side is to disable actor collision on the volume actor that hosts the rendered field. With collision turned off, the user can step inside the rendered volume and observe the electrostatic potential from within the plasma region, which we found useful for the spatial inspection of the field (see Fig. A.1 for a demonstration on the RGB-PVT scene).

The whole-tokamak scene in Fig. 3a used material settings chosen to expose the ridge and trough structure on the torus surface (albedo scale 0.18, black body scale 0.01, black body temperature 5000, density scale 0.1). Finally, the route exposed internal inspection through threshold filtering and material-driven clipping out of the box. These controls were not architecturally precluded by the PVT format, as community plug-ins, such as TBRaymarcherPlugin [47], demonstrated that clipping and transfer function controls can be authored directly into a custom PVT ray marcher, but they were not implemented in our shader graph PVT path, whereas they were surfaced as exposed material parameters of the SVT/HV pipeline.

A.IV. INTERACTIVE INSPECTION CONTROLS

A rendered volume that the user cannot enter, slice, or filter is of limited use for digital twin analysis, and the

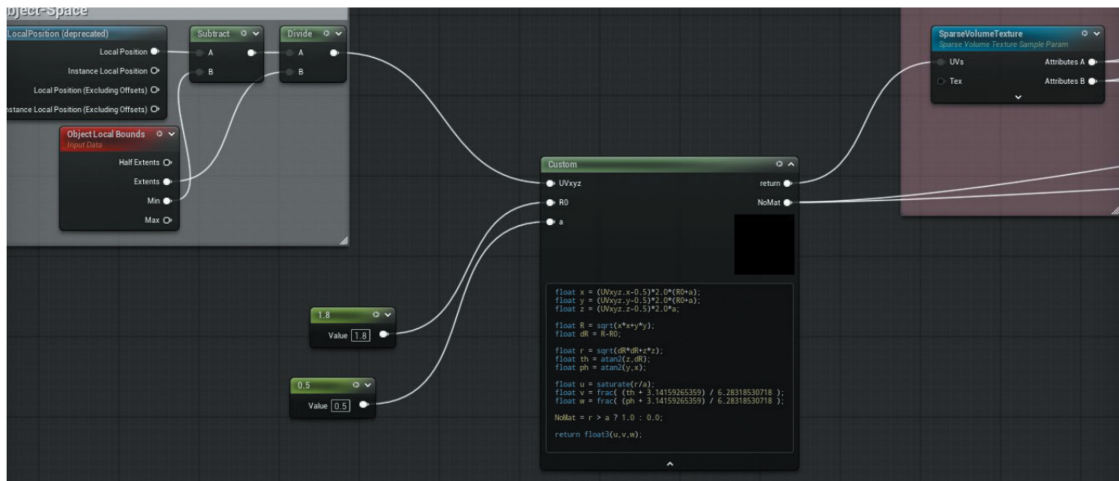


Fig. A.2. Excerpt of the SparseVolumeMaterial graph showing the HLSL custom node that implements the inverse map of Eq. (1). The center panel shows the HLSL source text.

APPENDIX B

OMNIVERSE IMPLEMENTATION AND OPERATION

This appendix documents the Omniverse-side implementation and operating procedures that support the approach in Sec. III and the demonstrations in Sec. IV. The extension, prim, and viewport identifiers quoted in this appendix were taken from Omniverse Kit SDK 109.0.1. As discussed in Sec. V, the in-engine demonstrations of the Omniverse paths were exercised in a separate environment from the Unreal-side measurements, without controlled per-frame instrumentation.

B.I. CARTESIAN OPENVDB PATHS AND USD STAGE ASSEMBLY

The OpenVDB-backed paths (a) and (b) in Table 1 followed the conversion pattern in Sec. III.B, with the Omniverse-side VDB assets produced as a parallel output at the voxel grid configurations listed in the table. For the gKPSP demonstration, the sequence consisted of 51 .vdb files, one per simulation snapshot, in which each file stored the normalized $|\phi|/\max|\phi|$ scalar in a single FloatGrid named density. The grid origin and voxel spacing were baked into each VDB through a linear transform aligned with the simulation (x, y, z) frame, so the file followed the same Cartesian OpenVDB convention as the input consumed by the Unreal Engine.

The full 51-frame sequence was wrapped into a single OpenUSD stage (see Fig. B.1). A UsdVol.

Volume prim was defined under the stage root, and a child OpenVDBAsset field referenced the VDB sequence through a time-sampled filePath attribute that resolved a per-snapshot .vdb file for each frame on a Z-up stage scaled to centimeter units. The stage-level timeCodesPerSecond attribute set the conversion between USD time codes and real-time seconds, while the advisory framesPerSecond attribute recorded the playback/display rate expected of the host applications.

In this demonstration, both were set to 24 fps, which let the USD Composer interpret each integer time code as one animation frame on its timeline, mirroring the role of the SVT animation block in Sec. IV.A.

B.II. RTX-SCIENTIFIC/INDEX CONFIGURATION AND INTERACTION

The setup was reproduced through three dialogs only. The user enables the `omni.index.kit.rtxscientific` extension (with companion compositor `omni.rtx.indexcomposite`) in the USD Composer's Extension Manager, switches the viewport renderer to RTX-Scientific, and opens one of the two OpenVDB-backed USD stages of the previous paragraph so that no additional asset import or rebake step is required (see Fig. B.2).

The identifiers and viewport labels in this section were taken from Omniverse Kit SDK 109.0.1; later Kit versions may rename or restructure these. Index additionally exposes a transfer function editor that maps the normalized magnitude $|\phi|/\max|\phi|$ stored in the VDB asset to color and opacity through a one-sided ramp (e.g., a black-to-orange amplitude scale).

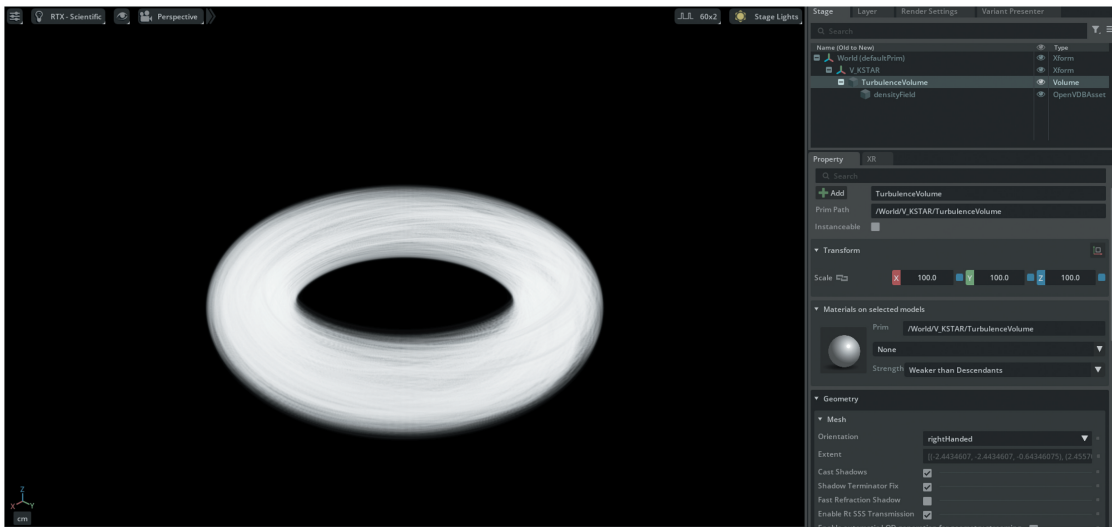


Fig. B.1. OpenUSD scene assembly. The gKPSP OpenVDB sequence is registered as a `UsdVol.Volume` under the stage root with its density field exposed as an `OpenVDBAsset`, rendered through the RTX-Scientific backend.

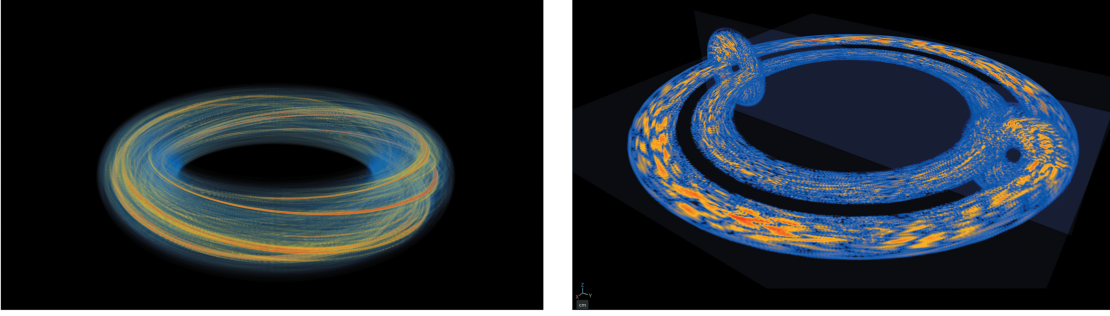


Fig. B.2. Omniverse RTX-Scientific (Index) interaction on the OpenVDB-backed gKPSP volume.

For still image capture, the same `UsdVol.Volume` prim can additionally be rendered with an `OmniVolumeDensity` material under RTX-Path Tracing, treated here as a complementary still rendering option rather than as the primary exploration mode because the convergence and sampling settings make it substantially slower to iterate than Index.

B.III. RAW-CONTAINER XAC PATH

Path (c) is the Index route. A converter script reformats the native float64 simulation output into a float32 C-order raw container in the layout that Index expects. Two implementation-critical steps, together with the periodic ghost plane convention recorded in Sec. III.C, make this conversion correct. First, the slightly nonuniform r -axis is linearly interpolated onto a uniform index axis using precomputed weights. Second, the result is serialized as little-endian float32 in C order. The resulting container has the shape $N_r \times (N_\theta + 1) \times (N_\phi + 1) = 256 \times 617 \times 193$, which is the same ghost-augmented grid that drives the SVT path in Sec. IV.A. A companion `meta.json` carries the global maximum $g_{\max} = \max |\phi|$ together with the geometric constants so that the USD scene authoring script and the XAC kernel share a single normalization and geometry source.

The corresponding USD scene distributes three roles across one stage. A `Volume` prim defines the placement of the raw field in centimeter-unit scene space and marks it as an Index-managed sparse volume. A `FieldAsset` selects Index’s raw float32 importer, an Omniverse/Index-specific extension to the OpenUSD `UsdVolVolume` field family rather than a portable schema, and binds the time-varying raw files to that volume. A `VolumeShader`, whose `info:xac:sourceCode` attribute contains the inverse toroidal transform and the color/opacity logic, carries the per-sample kernel. This attribute follows USD’s `info:<subsystem>:sourceCode` convention (see `info:mdl:sourceCode` for NVIDIA Material Definition Language (MDL) shaders), and the prim additionally sets `info:`

`implementationSource` to “sourceCode” so that `UsdShade` selects the inline source rather than a separately referenced `sourceAsset`.

Because the XAC variant is exposed by the `omni.index.kit.rtx_scientific` extension rather than by the core OpenUSD `UsdShade` schema, its identifier and Just-In-Time (JIT) semantics should be reverified when porting to a different Kit version or to another Index-based host.

The trade-off is that the executable XAC source still repeats several host-side constants (R_0 , a , $r_{\min}$, the volume scale and translation, and $g_{\max}$), which must be kept consistent with the host-side `xformOp` and the geometry record in `meta.json`. Host/device coordinate agreement, therefore, is a checked convention rather than a compile time invariant.

The complete kernel that implements the inverse transform alongside the color/opacity evaluation is reproduced in abridged form as Fig. B.3. This XAC path inherits Index’s transfer function editor, ambient occlusion, and multi-GPU scaling without modification. In the demonstration scene, the USD Composer Colormap editor was wired so that the kernel’s `lookup(t)` call used the color and opacity curve authored in the GUI directly, allowing the analyst to tune the positive/negative contrast, the opacity ramp, and the threshold window without recompiling the shader.

B.IV. KIT WARP EXTENSION

NVIDIA Warp [30] plays two distinct roles alongside the XAC route. It provides the GPU implementation of the toroidal to Cartesian resampling used in path (b), and it underlies a standalone ray marcher that implements the same render time inverse toroidal transform on the native $256 \times 616 \times 192$ grid that path (c) ultimately runs as $256 \times 617 \times 193$ inside Index. The standalone marcher is a development reference rather than a rendering deliverable.

The inverse map sign convention, the periodic wrap trilinear interpolation, and the handedness corrections noted


```

int execute(const Sample_info_self& s, Sample_output& o)
{
    // Object-space voxel coordinate -> scene meters
    const float3 P = s.sample_position_object_space;
    const float3 X = apply_volume_xform_in_metres(P);

    // Inverse toroidal transform (Y-up convention in this XAC path)
    const float Rcyl = sqrtf(X.x*X.x + X.z*X.z);
    float phi = wrap_2pi(atan2f(X.z, X.x));
    const float dR = Rcyl - R0;
    const float r = sqrtf(dR*dR + X.y*X.y);
    if (r > a || r < r_min) {
        o.set_color(make_float4(0,0,0,0));
        return NV_IDX_PROG_OK;
    }
    float theta = wrap_2pi(atan2f(X.y, dR));

    // Ghost-plane dimensions use (N_total - 1) on periodic axes.
    const float3 q = make_float3(
        phi / TWO_PI * (Nphi_g - 1.0f),
        theta / TWO_PI * (Ntheta_g - 1.0f),
        (r - r_min) / (a - r_min) * (Nr - 1.0f));

    const auto sampler = s.self.template generate_sampler<float>();
    const float scalar = sampler.fetch_sample(q);

    float sn = clamp(scalar / g_max, -1.0f, 1.0f);
    float4 c = s.self.get_colormap().lookup(0.5f + 0.5f * sn);
    c.w *= alpha0 + alpha1 * fabsf(sn) * fabsf(sn);
    o.set_color(c);
    return NV_IDX_PROG_OK;
}

```

Fig. B.3. Render time native grid sampling logic in the IndeX XAC path (abridged). The listing omits the boilerplate and helper definitions, but keeps the essential sequence of object space sample position, inverse toroidal map, native grid lookup, and transfer function evaluation.

in Sec. III.C are settled in this shorter Python + Warp compile and run cycle first, before the same logic is ported into the XAC kernel and reused in the Kit extension described next. The binary search on the nonuniform r -axis carries over to the Kit kernel of path (d) only. Because the standalone tool produces an external PNG rather than feeding the Composer viewport, it does not inherit the transfer function editor, cross section planes, or multi-GPU scaling of the IndeX backend, and its role, therefore, is restricted to development support.

Path (d) in Table 1 packages the same render time inverse toroidal transform as path (c), but instead of embedding it in IndeX, it ships as a custom Omniverse Kit extension. The extension follows the standard Kit layout (a configuration file declaring a dependency on omni.warp and omni.ui, plus a Python module) and is enabled from the Extension Manager in the same USD Composer session that hosts the OpenVDB and IndeX paths, with the motivation that the inverse transform pipeline becomes accessible to interactive control

authoring directly in Python rather than through XAC recompilation while keeping the same render time evaluation on the native (r, θ, ϕ) grid.

The extension is organized around a clean separation of concerns. An entry point layer subclasses omni.ext.IExt and instantiates the panel on startup, while a data access layer loads the nonuniform radial axis once at startup into a persistent GPU buffer and streams the current $256 \times 616 \times 192$ snapshot from disk into a re-allocated `wp.array3d` on each frame switch. The per-frame input/output cost is nontrivial and is discussed alongside the playback control in the following.

The compute layer defines a Warp ray-marching kernel that performs the per-sample inverse transform, the binary search on the nonuniform r -axis, periodic wrap trilinear interpolation, and front to back compositing, accompanied by a packing kernel that writes the floating-point RGBA buffer into the BGRA8 format expected by omni.ui.ByteImageProvider. A separate User Interface (UI) layer assembles an

`omni.ui.Window`, drives the kernels via `wp.launch`, and pushes the resulting bytes into the panel.

The panel exposes the controls that an analyst typically needs while inspecting a turbulent potential frame. Time-dependent playback through a frame combobox, Play/Pause toggle, and playback rate slider iterates over the full 51-frame sequence, streaming the selected snapshot from disk into the `wp.array3d` GPU buffer that the kernel then ray marches. Camera-view presets (3/4 view, top, and side) provide reproducible figure framings for capture and comparison. An axis-aligned slice combobox selects no slab, a X -, Y -, or Z -aligned slab with normalized position and thickness sliders, with the slab test performed inside `ray_march_native` so that samples outside the slab are skipped and in-slab samples receive an alpha boost [emulating the visual effect of the “Create Slice” workflow that a USD Composer user expects from the OpenVDB paths, but implemented entirely inside the Warp kernel and applied to the native (r, θ, ϕ) field].

Finally, three `omni.ui.ColorWidgets` plus three scalar sliders set the negative, midpoint, and positive color stops of a parametric diverging color map together with an alpha baseline α_0 , a density coupled gain α_1 , and

a density emphasis exponent γ , so that the per-sample opacity follows $\alpha_0 + \alpha_1 |s_{\text{norm}}|^\gamma$, where $s_{\text{norm}} = \phi/g_{\text{max}}$, with γ above unity isolating the high-amplitude eddies in the same sense as the threshold filter in the previous RTX-Scientific paragraph, but as a smoothly tunable control rather than a hard window.

Rendering into a panel rather than into the Composer viewport (Fig. B.4) is a deliberate consequence of treating Kit as the host process. The trade-off is that path (d) does not inherit Index’s transfer function editor, cross-section plane primitives, or multi-GPU scaling. In exchange, it gains unrestricted Python authoring of the controls listed previously and time-dependent playback in the current Kit extension. Because the Kit kernel follows the same per-sample logic as the standalone Warp marcher, the inverse transform behavior observed during standalone development carries over to path (d) without further porting.

Together with the XAC path of path (c), path (d) demonstrates that the render time inverse toroidal transform of Eq. (1) can be deployed inside Omniverse through two complementary mechanisms, inside the Index backend through XAC and inside the Kit process through Warp.

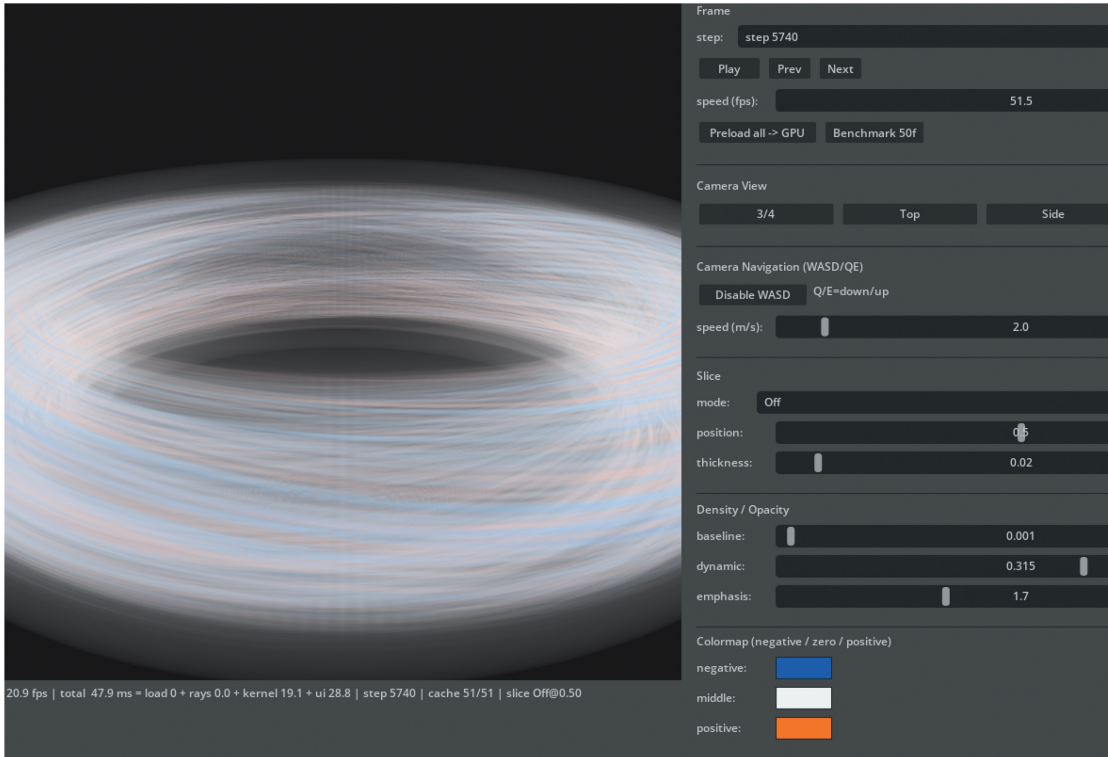


Fig. B.4. In-Kit Warp panel of path (d), captured on the RTX 3090 Ti workstation with the optimized kernel. The extension renders the native (r, θ, ϕ) field into an `omni.Ui.window` inside the USD Composer and exposes frame playback, camera presets, slice controls, density/opacity parameters, and a three-color diverging map. The status bar of this capture reads 20.9 fps at the depicted frame (47.9 ms in total, of which 19.1 ms is the ray-marching kernel and 28.8 ms is the `omni.Ui` image update).

Acknowledgments

The authors gratefully acknowledge Professor Mark S. Shephard of the SCOREC at RPI and Mark Beall of Simmetrix, Inc. for their generous software support throughout this work. We also thank Jinseop Park and Juhyung Kim at the Korea Institute of Fusion Energy for their assistance with access to the KAIROS supercomputer used in the present demonstrations.

The authors contributed equally to this work.

Author Contributions

CRedit: **Darian Figuera-Michal:** Data curation, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft; **Eui-Hyun Jung:** Investigation, Methodology, Software, Visualization, Writing – original draft, Writing – review & editing; **Mintae Na:** Conceptualization, Investigation, Methodology; **Sun-Jeong Kim:** Conceptualization, Investigation, Project administration, Supervision; **Jong-Hyun Kim:** Conceptualization, Methodology, Writing – review & editing; **Jung Lee:** Conceptualization, Methodology, Writing – review & editing; **Yu-Seop Kim:** Conceptualization, Methodology; **Jungpyo Lee:** Conceptualization, Methodology; **Jaemin Seo:** Conceptualization, Methodology; **Il-Hwan Kim:** Data curation, Investigation, Resources; **Taeuk Moon:** Conceptualization, Investigation, Methodology, Visualization; **Jae-Min Kwon:** Conceptualization, Methodology, Resources, Supervision; **Eisung Yoon:** Conceptualization, Funding acquisition, Project administration, Resources, Supervision, Writing – review & editing.

Disclosure Statement

No potential conflict of interest was reported by the author(s).

During the preparation of this work the authors used Claude Code Opus4.7 in order to improve the language and readability of the paper. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the paper.

Funding

This research was supported by the National Research Foundation of Korea grant funded by the Korea government [grant no. RS-2023-00254695].

ORCID

Eisung Yoon  <http://orcid.org/0000-0001-7107-7302>

References

1. M. I. Battye and S. Perinpanayagam, “Digital Twins in Fusion Energy Research: Current State and Future Directions,” *IEEE Access*, **13**, 75787 (2025); <https://doi.org/10.1109/ACCESS.2025.3561920>.
2. N. Bhatia et al., “Advanced Techniques for Fusion Data Visualisation,” *Front. Phys.*, **13**, 1569248 (2025); <https://doi.org/10.3389/fphy.2025.1569248>.
3. J.-M. Kwon et al., “Development of a Virtual Tokamak Platform,” *Fusion Eng. Des.*, **184**, 113281 (2022); <https://doi.org/10.1016/j.fusengdes.2022.113281>.
4. J.-M. Kwon et al., “Progress in Digital Twin Development of Virtual Tokamak Platform,” *IEEE Trans. Plasma Sci.*, **52**, 9, 3910 (2024); <https://doi.org/10.1109/TPS.2024.3390159>.
5. D. A. Ibanez et al., “PUMI: Parallel Unstructured Mesh Infrastructure,” *ACM Trans. Math. Softw.*, **42**, 3, 1 (2016); <https://doi.org/10.1145/2814935>.
6. G. Jo et al., “Development of a Gyrokinetic Hyperbolic Solver Based on Discontinuous Galerkin Method in Tokamak Geometry,” *Comput. Phys. Commun.*, **273**, 108265 (2022); <https://doi.org/10.1016/j.cpc.2021.108265>.
7. M. Krüger et al., “Game Engines for Immersive Visualization: Using Unreal Engine Beyond Entertainment,” *PRESENCE*, **33**, 31 (2024); https://doi.org/10.1162/pres_a_00416.
8. M. Schlüter et al., “Rendering Large Volume Datasets in Unreal Engine 5: A Survey,” *arXiv Preprint arXiv:arXiv:2504.07485* (2025); <https://arxiv.org/abs/2504.07485>.
9. R. Bluff et al., “ILM Presents ‘This is the Way’: The Making of Mandalorian,” *ACM SIGGRAPH 2020 Production Sessions, SIGGRAPH ’20*, Association for Computing Machinery, New York, NY (2020); <https://doi.org/10.1145/3368850.3383439>.
10. N. Ohno et al., “Digital-LHD: LHD Visualizer for Head-Mounted Display Systems,” *Plasma Fusion Res.*, **19**, 1401029 (2024); <https://doi.org/10.1585/pfr.1401029>.
11. N. Ohno and A. Kageyama, “VOIR: Virtual Reality Visualization Software for Large-Scale Simulations,” *Plasma Fusion Res.*, **19**, 1401024 (2024); <https://doi.org/10.1585/pfr.19.1401024>.
12. N. Bhatia et al., “Visualizing Digital Twins of Fusion Power Plants Using NVIDIA Omniverse,” *AIP Adv.*, **15**, 4, 045018 (2025); <https://doi.org/10.1063/5.0261883>.
13. S. Gazzotti et al., “Virtual and Augmented Reality Use Cases for Fusion Design Engineering,” *Fusion Eng. Des.*, **172**, 112780 (2021); <https://doi.org/10.1016/j.fusengdes.2021.112780>.

14. D. Iglesias et al., “Digital Twin Applications for the JET Divertor,” *Fusion Eng. Des.*, **125**, 71 (2017); <https://doi.org/10.1016/j.fusengdes.2017.10.012>.
15. “Sparse Volume Textures in Unreal Engine,” Unreal Engine 5 Documentation, Epic Games, Cary, NC (2024); <https://dev.epicgames.com/documentation/en-us/unreal-engine/sparse-volume-textures-in-unreal-engine>.
16. “Heterogeneous Volumes in Unreal Engine,” Unreal Engine 5 Documentation, Epic Games, Cary, NC (2024); <https://dev.epicgames.com/documentation/en-us/unreal-engine/heterogeneous-volumes-in-unreal-engine>.
17. K. Museth, “VDB: High-Resolution Sparse Volumes with Dynamic Topology,” *ACM Trans. Graph.*, **32**, 3, 1 (2013); <https://doi.org/10.1145/2487228.2487235>.
18. K. Museth, “NanoVDB: A GPU-Friendly and Portable VDB Data Structure for Real-Time Rendering and Simulation,” *ACM SIGGRAPH 2021 Talks, SIGGRAPH*, Association for Computing Machinery, New York, NY (2021); <https://doi.org/10.1145/3450623.3464653>.
19. J.-M. Kwon et al., “ITG-TEM Turbulence Simulation with Bounce-Averaged Kinetic Electrons in Tokamak Geometry,” *Comput. Phys. Commun.*, **215**, 81 (2017); <https://doi.org/10.1016/j.cpc.2017.02.009>.
20. G. Jo et al., “A Field-Aligned Gyrokinetic Solver Based on Discontinuous Galerkin in Tokamak Geometry,” *Comput. Phys. Commun.*, **316**, 109769 (2025); <https://doi.org/10.1016/j.cpc.2025.109769>.
21. “Mass Entity in Unreal Engine,” Unreal Engine 5 Documentation, Epic Games, Cary, NC (2024); <https://dev.epicgames.com/documentation/en-us/unreal-engine/mass-entity-in-unreal-engine>.
22. B. Karis, R. Stubbe, and G. Wihlidal, “A Deep Dive into Nanite Virtualized Geometry,” *ACM SIGGRAPH 2021, Advances in Real-Time Rendering in Games (Course Notes)* (2021); https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf.
23. R. Brucks, “Creating a Volumetric Ray Marcher,” *Shader Bits blog* (2016); <https://shaderbits.com/blog/creating-volumetric-ray-marcher>.
24. I. Wald et al., “Embree: A Kernel Framework for Efficient CPU Ray Tracing,” *ACM Trans. Graph.*, **33**, 4, 1 (2014); <https://doi.org/10.1145/2601097.2601199>.
25. J. Sarton et al., “State-of-the-Art in Large-Scale Volume Visualization Beyond Structured Data,” *Comput. Graphics Forum*, **42**, 3, 491 (2023); <https://doi.org/10.1111/cgf.14857>.
26. T. Moon et al., “Development of Novel Collision Detection Algorithms for the Estimation of Fast Ion Losses in Tokamak Fusion Device,” *Comput. Phys. Commun.*, **309**, 109490 (2025); <https://doi.org/10.1016/j.cpc.2024.109490>.
27. G. Diamond et al., “PUMIPic: A Mesh-Based Approach to Unstructured Mesh Particle-in-Cell on GPUs,” *J. Parallel Distrib. Comput.*, **157**, 1 (2021); <https://doi.org/10.1016/j.jpdc.2021.06.004>.
28. “Customized UVs in Unreal Engine Materials,” Unreal Engine 5 Documentation, Epic Games, Cary, NC (2024); <https://dev.epicgames.com/documentation/en-us/unreal-engine/customized-uvs-in-unreal-engine-materials>.
29. “FMeshDescription API Reference,” Unreal Engine 5 Documentation, Epic Games, Cary, NC (2023); <https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/MeshDescription/FMeshDescription>.
30. “NVIDIA Warp: A Python Framework for High-Performance GPU Simulation and Graphics,” NVIDIA Corporation, GitHub (2024); <https://github.com/NVIDIA/warp>.
31. “NVIDIA IndeX 3D Volumetric Visualization Framework,” NVIDIA Corporation, Santa Clara, CA (2024); <https://developer.nvidia.com/nvidia-index/>.
32. “NVIDIA IndeX XAC (Xtreme Accelerated Computing) Programmable Sampling Stage,” NVIDIA IndeX Documentation, NVIDIA Corporation, Santa Clara, CA (2026).
33. “RTX – Scientific (IndeX) Renderer in Omniverse,” NVIDIA Omniverse Materials and Rendering Documentation NVIDIA Corporation, Santa Clara, CA (2024); https://docs.omniverse.nvidia.com/materials-and-rendering/latest/rtx_scientific_index.html.
34. M. Levoy, “Display of Surfaces from Volume Data,” *IEEE Comput. Graphics Appl.*, **8**, 3, 29 (1988); <https://doi.org/10.1109/38.511>.
35. K. Engel, M. Kraus, and T. Ertl, “High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading,” *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware (EGGH01)*, p. 9, Eurographics Association, Eindhoven, The Netherlands (2001); <https://doi.org/10.2312/EGGH/EGGH01/009-017>.
36. C. Lee et al., “Scalable Modeling of 3D Eddy Current Problem for Magnetic Fusion Devices,” *Comput. Phys. Commun.*, **317**, 109864 (2025); <https://doi.org/10.1016/j.cpc.2025.109864>.
37. T. Rhee et al., “Numerical Study of Neutral Beam Injection Prompt Loss at KSTAR Poloidal Limiters,” *Phys. Plasmas*, **26**, 11, 112504 (2019); <https://doi.org/10.1063/1.5117330>.
38. K. Kim et al., “ δf Monte Carlo Calculation of Neoclassical Transport in Perturbed Tokamaks,” *Phys. Plasmas*, **19**, 8, 082503 (2012); <https://doi.org/10.1063/1.4740511>.
39. E. S. Seol and M. S. Shephard, “Efficient Distributed Mesh Data Structure for Parallel Automated Adaptive Analysis,” *Eng. Comput.*, **22**, 3–4, 197 (2006); <https://doi.org/10.1007/s00366-006-0048-4>.
40. C. Geuzaine and J.-F. Remacle, “Gmsh: A 3-D Finite Element Mesh Generator with Built-In Pre- and Post-Processing Facilities,” *Int. J. Numer. Methods Eng.*, **79**, 11, 1309 (2009); <https://doi.org/10.1002/nme.2579>.

41. “SimModeler: Simulation Modeling Suite,” Simmetrix, Inc., Clifton Park, NY (2024); <https://www.simmetrix.com/>.
42. CadQuery Contributors, “CadQuery: A Python Parametric CAD Scripting Framework Based on OCCT,” GitHub (2024); <https://github.com/CadQuery/cadquery>.
43. Open Cascade SAS, “Open CASCADE Technology (OCCT),” Open Cascade SAS, Guyancourt, France (2025); <https://dev.opencascade.org/>.
44. J. S. Merson et al., “PCMS: Parallel Coupler for Multimodel Simulations,” *arXiv Preprint arXiv: arXiv:2510.18838* (2025); <https://arxiv.org/abs/2510.18838>.
45. “Rendering to Multiple Displays with nDisplay in Unreal Engine,” Unreal Engine 5 Documentation, Epic Games, Cary, NC (2024); <https://dev.epicgames.com/documentation/en-us/unreal-engine/rendering-to-multiple-displays-with-ndisplay-in-unreal-engine>.
46. Lee C, Kim J, Jo G, Kwon J, Bak J and Lee J. (2025). Scalable modeling of 3D eddy current problem for magnetic fusion devices. *Computer Physics Communications*, 317 109864 [10.1016/j.cpc.2025.109864](https://doi.org/10.1016/j.cpc.2025.109864)
47. T. Bazar, “TBRaymarcherPlugin: DICOM/MHD Volume Raymarcher for Unreal Engine,” GitHub (2023); <https://github.com/tommybazar/TBRaymarcherPlugin>.