Riley: A computational framework for higher-order finite element image synthesis applied to digital image correlation uncertainty quantification

Lloyd Fletcher[1,*], Joel Hirst[1,†], Wiera Bielajewa[2,‡]
[1]UK Atomic Energy Authority, UKAEA South Yorkshire, Rotherham, UK
[2]UK Atomic Energy Authority, Culham Campus, Culham, UK
[*]Corresponding author: lloyd.fletcher@ukaea.uk, ORCID: 0000-0003-2841-8030
[†]ORCID: 0000-0001-6041-0106
[‡]ORCID: 0009-0004-0009-1077

Abstract

Image-based measurements such as digital image correlation (DIC) have the potential to reduce the cost and risk associated with validation experiments for high-value components in fusion engineering. To use DIC effectively in this setting, we need to design experiments computationally before physically realising them, including the effects of camera placement, image formation, finite-element deformation, and measurement uncertainty.

We present `Riley`, a dependency-free software rasteriser written in Zig for creation of synthetic DIC images directly from deformed finite-element surface meshes. `Riley` supports rendering linear triangular `tri3`, quadratic triangular `tri6`, linear qaudrilateral `quad4`, and quadratic quadrilateral `quad8`/`quad9` elements. `Riley` also supports mixed-mesh scenes, multiple cameras, analytic and nodal shaders, and higher-order texture sampling. Our method uses inverse-mapped rasterisation, projected-residual Newton solves for higher-order elements, adaptive hulls for raster-space rejection and tile assignment, compile-time kernel specialisation, and explicit single instruction multiple data (SIMD) execution.

We verify our renderer using targeted tests of inverse-mapping reprojection error, silhouette coverage, pixel sub-sampling convergence, and depth-buffer visibility. The Newton pathway gives reprojection errors below 10^{-12} px for regular and affine-shear cases and below approximately 10^{-10} px for challenging bulge cases. Rendered single-element silhouettes match independently generated reference masks exactly, with area errors no more than $3 \times 10^{-3}\%$. Performance benchmarks show single-threaded raster throughput of 242 MPx/s for the direct `tri3` nodal pathway, and adaptive hulls combined with SIMD provide up to 12.99× raster-loop speedup. Thread-scaling analysis yields fitted parallel fractions of 99.0% for in-memory and 98.8% for disk-output workflows. These results show that `Riley` provides a verified and performant foundation for large-scale DIC uncertainty quantification and simulation-driven experimental design.

Keywords: digital image correlation, uncertainty quantification, rasterisation, synthetic image generation

Article Highlights

- We developed the first offline rasteriser specifically for digital image correlation uncertainty quantification supporting higher-order elements, texture sampling, and camera distortion
- We created an adaptive hull algorithm and a projected residual Newton solver that makes higher-order element rasterisation computationally efficient
- Our Zig implementation leverages explicit single-instruction-multiple-data execution and parallel scaling to significantly accelerate representative digital image correlation uncertainty quantification workflows

1 Introduction

1.1 Background and motivation

In high-value engineering applications such as nuclear fusion there is a need to make risk-aware decisions based on multi-physics simulations with uncertainty quantification [1–5]. This need stems from the requirement for components to withstand extreme environments that are not always testable before the component goes into service. De-risking and reducing simulation uncertainty requires experimental validation; however, the testable domain for fusion is limited and large component-scale experiments can be expensive and time consuming [6–11]. To address this, we need to extract as much useful information as possible from the experiments we can perform in order to populate as much of the validation domain as possible at minimum cost. One way to increase the information gained from a validation experiment is to deploy data-rich image-based sensors such as digital image correlation (DIC). Optimising the deployment of image-based sensors, and mitigating uncertainty before physically deploying them, has the potential to reduce the cost of validation experiments and reduce the occurrence of failed experiments due to sensor failure or inadequacy.

DIC is a non-contact, image-based technique for extracting the shape and surface displacements from a sample that is imaged during deformation [12, 13]. For DIC measurements a random pattern of black and white speckles is applied to the surface of the sample. Tracking this pattern with a single camera provides 2D measurements of surface displacement, whereas with a two-camera stereo-DIC system the sample shape and the three components of the surface displacement can be obtained. Using the displacement fields obtained from DIC, the strain fields can be numerically derived by spatial differentiation. DIC is beneficial for fusion-relevant testing, where experiments may involve high heat fluxes, vacuum environments, electromagnetic constraints, and limited direct access to the component [6, 9, 11, 14], because we can place the cameras outside the test chamber. Furthermore, as thermal strains are often of interest in fusion applications, a non-contact technique is essential to separate sensor thermal expansion from that of the component, which is a common challenge with strain gauges. In addition to these benefits, DIC also provides a wealth of data that is not readily available with point sensors. A key example is the ability to measure boundary conditions or geometric arrangements of components that can have a significant effect on the resulting physical response of a component. For example, DIC and related full-field measurements have been used to support finite-element model validation and mechanics interpretation from experimental displacement fields [15–19]. In these cases, if DIC was not available, differences in boundary conditions or geometric arrangement could be misattributed as a fundamental difference in the physics between the simulation and experiment, reducing confidence in using the simulation to make decisions about component performance.

In order to use experimental data for simulation validation, we must quantify the uncertainty of the experimental measurement [18–21]. There are two main types of measurement error: random and systematic errors. The main source of random error in the DIC measurement chain is electrical noise in the camera sensor itself. This noise is typically heteroscedastic and dependent on the grey level. We typically characterise random errors for DIC using statistical methods by taking a series of images under steady-state conditions and analysing the central tendency and dispersion of the quantity of interest [13]. We find that systematic errors in the DIC measurement chain are more difficult to characterise and typically require image-rendering simulation methods to estimate [22–25]. In practice, we can achieve a measurement resolution of approximately 0.01 pixels for the in-plane displacement components and 0.1 pixels for the out-of-plane displacement with a stereo-DIC system [12, 13]. The main sources of systematic errors in the DIC measurement chain include spatial integration and discretisation, temporal discretisation and integration, image digitisation, image dynamic range and contrast, missing data at the edges requiring reconstruction, and stereo-calibration errors when using a stereo-DIC system [26–30]. The trade-off between random and systematic errors in the DIC measurement chain is directly linked to the selection of the processing parameters used to analyse the images. For example, in subset-based DIC, to stay within the Nyquist limit the minimum sized subset typically needs speckles of approximately 3 pixels with 3 transitions from black to white across the subset, leading to a typical recommended minimum subset size of 12×12 pixels [13]. However, a

user can specify a much larger subset than this. The trade-off is that a larger subset will reduce the random error while spatially smearing gradients in the displacement field, increasing the systematic error [23, 31]. Selecting the optimal processing parameters is test dependent, as the trade-off between random and systematic error is linked to the camera parameters and how these sample the gradients in the displacement and strain fields.

In order to characterise the trade-offs between DIC processing parameters and the associated measurement errors, researchers have developed synthetic image deformation techniques [22, 25, 32, 33]. These methods provide a controlled route for studying how image formation, rendering assumptions, and DIC processing choices influence the resulting measurement uncertainty. In the context of fusion validation this is particularly important, because the cost of large-scale experiments means that the design space associated with camera placement, image quality, optical setup, and DIC processing cannot be explored efficiently by physical testing alone. Synthetic image generation is therefore not only a tool for uncertainty analysis, but also a tool for simulation-driven experimental design [3, 4].

Therefore, there is a clear need for robust and performant computational frameworks for image synthesis from finite element simulations for DIC UQ. Such frameworks must be accurate enough to study systematic errors at sub-pixel scales, while also being computationally efficient enough to make large offline rendering sweeps practical for uncertainty quantification and experimental design. This need sits at the intersection of image-based experimental mechanics, uncertainty-aware test design [16, 17, 34, 35], and computational frameworks that preserve geometric information through simulation pipelines [36, 37].

1.2 Limitations of existing approaches

Solid mechanics finite element (FE) simulations often employ quadratic or higher-order elements, as these can provide an improved tradeoff between degrees of freedom and solution accuracy [38–40]. This is contrasted against graphics pipelines for simulation visualisation where there is a strong need for real-time rendering. These visualisation pipelines are typically optimised for linear triangles and therefore rely on adaptive sub-tessellation of higher-order elements. A key example of this is the Visualization Toolkit (VTK) [41], which uses tessellation of higher-order finite elements into linear cells for conventional visualisation pipelines [42]. VTK uses image-order rasterisation which is specifically designed for fast rendering of linear triangles by leveraging modern graphics hardware. Exceptions to sub-tessellation approches include ray-casting for curved elements [43], adaptive high-order FE visualisation methods [44], pixel-exact rendering approaches [45], and ELVis [46, 47]. However, these tools are focused on accurate or interactive visualisation of high-order FE fields rather than synthetic image generation for DIC UQ, and therefore do not address the specific requirements of rendering deformed speckle patterns with controlled interpolation and image-formation fidelity. These visualisation tools also have distinctly different requirements to DIC UQ in terms of accuracy versus real-time rendering needs. For visualisation purposes the tradeoff favours real-time rendering, whereas for DIC UQ rendering can be performed offline and accuracy is prioritised due to the sub-pixel resolution of DIC. A further distinction is that conventional FE visualisation pipelines are generally designed around idealised viewing models and do not explicitly incorporate calibrated camera distortion as part of the rendering formulation. This is a significant limitation for DIC UQ, where the image-formation model should reflect the optical system used in the measurement chain, including lens distortion and camera intrinsics [22, 32].

Visualisation graphics pipelines commonly make extensive use of single-precision floating point arithmetic, whereas the accuracy requirements of DIC UQ favour double precision. DIC is also commonly performed using monochrome cameras, as these provide a direct, un-interpolated measurement of light intensity at every pixel, helping preserve sharp speckle gradients for sub-pixel displacement tracking [12, 13]. We note here that low triangle-count sub-tessellation of higher-order finite elements can introduce parasitic rendering errors that may then be folded into the apparent systematic error of the DIC measurement chain. Recommendations for the level of sub-tessellation required for different realistic rendering scenarios to overcome this parasitic bias for DIC UQ remain an open research question.

While real-time rendering is not essential for DIC UQ, performance is still important, as exper-

imental design optimisation sweeps and UQ for inverse methods can require large parameter studies and Monte Carlo sampling [3, 4, 24]. Given that DIC UQ allows for offline rendering, we can exploit new strategies for accelerating rendering computations with parallelisation, where both inter- and intra-image parallelisation are possible because frame ordering does not matter. Typically a user will provide a complete finite element simulation from which they wish to render a set of simulated deformed speckle pattern images. In this case the order of arrival of images is not important, as the user wants the full stack of frames as fast as possible. DIC UQ also involves multi-camera rendering for stereo camera simulation [22, 32] or multi-camera setups [29], and sensor sizes are generally larger than a typical screen used for interactive visualisation. Many DIC workflows use multi-megapixel machine-vision cameras, and higher-resolution setups of 24 MPx or more are increasingly practical [14]. For multi-camera rendering it is also possible to parallelise the computation over cameras rendering the same scene.

Synthetic image deformation methods for 2D DIC have been developed to study DIC error sources and image-registration bias [23, 31, 48] and were then extended to stereo DIC by Balcaen et al. [22, 32]. In contrast to conventional FE visualisation pipelines, Balcaen et al. explicitly include calibrated stereo camera projection with intrinsic and extrinsic parameters together with lens distortion, thereby tying the synthetic images to the optical model of the measurement system rather than to an idealised undistorted camera. For 2D DIC there are also analytic methods, such as those developed by Sur et al. [48]. These analytic methods have provided an important baseline for comparing the measurement resolution of different DIC codes in the DIC Challenge [49, 50]. Bornert et al. [26] and Rossi et al. [23] showed that texture interpolation used for synthetic image deformation can lead to significant bias, particularly when the same interpolation model is used in both image generation and DIC processing. Rossi et al. [23] also investigated how increasing the number of sub-pixel samples and averaging can reduce this type of image-generation bias. The method developed by Balcaen et al. relies on projecting the FE kinematics into image space and then using this projected mesh to warp an existing 2D reference image. This has the benefit that real experimental reference images can be used, thereby carrying through the exact signature of the speckle pattern used for the test. However, the method remains fundamentally an image-space deformation approach in which the main remaining numerical error source is the interpolation of the reference image, and it is less naturally suited to pre-test experimental design where no physical reference image yet exists. Rohe and Jones [33] also investigated the use of the open-source graphics tool Blender, demonstrating comparable synthetic-image accuracy to the image deformation tool implemented in the commercial DIC software MatchID for the cases they considered. Rohe and Jones used Blender's Cycles rendering engine, which is a physically based path tracer, and concluded that while it can achieve comparable accuracy it does so at a greater computational cost. At the same time, the use of a path tracer brings significant advantages that remain under-explored in the DIC literature, including more realistic treatment of lighting, depth of field, material appearance, reflections and refractions, together with the ability to simulate light interacting with the scene rather than simply deforming an existing image. This is particularly important for more complex optical scenarios, where effects such as specular reflections, light transmission through transparent media, or imaging through distorting interfaces such as vacuum chamber windows or fluid layers may become important. However, the Blender workflow presented by Rohe and Jones operates through Blender's conventional triangle-based surface-mesh representation and does not discuss the direct rendering of higher-order finite elements or the effect of higher-order geometric fidelity on DIC UQ. In contrast to the object-order path tracing approach used in Blender the use of image-order rasterisation rendering is completely un-explored for application to DIC UQ. While rasterisation does not support the same level of optical fidelity as path tracing it has the potential to be significanly faster while maintaing an suitable level of accuracy that is required for DIC experimental design and UQ.

Therefore, there is a need for a computational rasterisation framework specifically designed for DIC UQ that supports higher-order elements and higher-order texture sampling. Such a framework should provide a balance between accuracy and computational performance, while operating directly on the deformed 3D scene rather than only through projected image warping. This is particularly important for DIC UQ and simulation-driven experimental design, where large offline rendering sweeps may be

required to investigate camera placement, sensor configuration, and measurement uncertainty before any physical test is performed [3, 4, 34, 35].

1.3 Contributions of this work

In this work, we present `Riley`, a computational framework and software rasteriser designed for accurate image synthesis from higher-order finite elements for digital image correlation uncertainty quantification (DIC UQ). The framework combines a set of algorithms chosen to balance the rendering accuracy required for DIC UQ against the computational performance needed for experimental design sweeps. In particular, we target the coupled challenges of rendering deformed higher-order finite element geometry and synthesising speckle images with higher-order texture sampling. The proposed framework is released as fully open-source software under the MIT licence, enabling transparent adoption, reproducibility, and extension by the community [51]. The specific contributions of this work are:

- Robust rasterisation methods for linear and higher-order finite elements in the same scene, including a novel inverse-mapping treatment of nonlinearly projected elements using an image-order rasterisation technique.
- Efficient sensor point early-rejection strategies for higher-order finite elements using "adaptive hulls", reducing the number of expensive inverse mapping iterations required during rasterisation.
- Extension of the rasterisation framework beyond the ideal pinhole camera to include camera intrinsics and Brown–Conrady lens distortion, a capability that is uncommon in software rasterisers but essential for physically realistic DIC UQ image synthesis.
- Implementation of higher-order texture sampling methods for speckle image synthesis, enabling accurate image formation for DIC UQ.
- A practical software implementation in the Zig programming language, using compile-time specialisation, branch minimisation, and explicit single instruction multiple data (SIMD) vector types for performance.
- A hierarchical offline rendering parallelisation model spanning both intra-image and inter-image workloads with scaling demonstrated on a realistic DIC UQ test case.

We begin the paper by outlining the problem definition, including the inputs and outputs of the rasteriser and the associated design requirements for DIC UQ. We then describe the computational methodology, giving an overview of the rendering pipeline, early rejection methods, inverse mapping of higher-order elements, and shading strategies with nodal fields or textures. Following this, we describe the software implementation of the computational method, including the tiling architecture, the compile-time kernels, and the use of SIMD. Finally, we present verification studies, computational performance results, and a capability demonstration for synthetic image generation in DIC UQ workflows.

2 Problem definition & mathematical framework

The synthetic image deformation process for DIC UQ takes one or more solid mechanics finite element (FE) simulations together with a set of user-specified cameras and renders a sequence of deformed images which we can then analyse with DIC. The FE input consists of nodal coordinates, element connectivity, and nodal displacement fields as a function of time. For texture-based image synthesis, a speckle-pattern texture and corresponding nodal mapping coordinates (often called uv's in computer graphics) are also required.

In practical DIC UQ studies, a scene may contain multiple deforming or static meshes with different element types and different shading strategies. An example is the study in [11], where the relative

position of an induction coil was tracked relative to the heated sample. As DIC is a surface measurement technique, we only need to render surface element meshes, and these may be pre-extracted from a three-dimensional simulation to reduce computational cost. In the present work, our rasteriser supports the surface element types linear triangles (tri3), quadratic triangles (tri6), linear quadrilaterals (quad4), and quadratic quadrilaterals (quad8 and quad9).

The output of the synthetic image deformation process is a sequence of raster images per camera, with one image produced for each requested frame and camera combination. These images form the input to the downstream DIC analysis. This defines the scope of the rendering problem considered here: given deformed FE surface geometry, camera models, and shading data, our aim is to compute physically consistent synthetic images for offline DIC UQ studies. Our main design requirements for `Riley` are therefore:

- direct rendering from deformed three-dimensional FE surface meshes,
- support for linear and higher-order surface elements, including tri3, tri6, quad4, quad8, and quad9,
- support for multiple meshes, element types, and shading strategies within a single scene,
- support for calibrated camera models and sub-sample anti-aliasing consistent with the accuracy requirements of DIC UQ,
- support for higher-order texture sampling for speckle image synthesis,
- sufficient computational performance for offline rendering studies, including multi-camera and multi-frame parameter sweeps.

2.1 Image synthesis from finite element simulations

We now build the mathematical framework for `Riley`, beginning with the camera projection and visibility problem for rasterisation. Let $\mathbf{x}_e : \hat{\Omega}_e \to \mathbb{R}^3$ denote the geometry map of surface element e, which maps parent coordinates $\boldsymbol{\xi}$ in the domain $\hat{\Omega}_e$ to world coordinates, and let π denote the camera projection function. For a sensor point at $\mathbf{p}$, rasterisation requires the determination of a parent coordinate $\boldsymbol{\xi}$ through the forward mapping such that

$$\pi(\mathbf{x}_e(\boldsymbol{\xi})) = \mathbf{p}. \tag{1}$$

This defines the inverse-mapping problem

$$\mathbf{R}_e(\boldsymbol{\xi}, \mathbf{p}) = \pi(\mathbf{x}_e(\boldsymbol{\xi})) - \mathbf{p} = \mathbf{0}, \qquad \boldsymbol{\xi} \in \hat{\Omega}_e, \tag{2}$$

where $\mathbf{R}_e(\boldsymbol{\xi}, \mathbf{p})$ is the projection residual function. If multiple elements in the scene satisfy this constraint, the visible solution is the one that lies in front of the sensor plane and is closest to the sensor plane. For a visible sensor point at $\mathbf{p}$, with corresponding parent coordinate $\boldsymbol{\xi}$ on element e, we evaluate the intensity through the shader function s_e of element e as

$$I_e(\mathbf{p}) = s_e(\boldsymbol{\xi}), \tag{3}$$

where I_e is the sensor-point intensity function induced by element e.

We perform sensor-point evaluation at the sub-pixel level in order to apply sub-sample anti-aliasing. Let $\mathbf{p}_{ij}^{(s_x,s_y)}$, with $s_x = 0, \ldots, S_x - 1$ and $s_y = 0, \ldots, S_y - 1$, denote the sub-pixel sensor-point locations of output pixel (i, j), where S_x and S_y are the numbers of sub-pixel points in the two raster directions and $S = S_x S_y$ is the total number of sub-pixel points per pixel. The resolved pixel intensity I_{ij} is then obtained by averaging the corresponding sub-pixel intensities,

$$I_{ij} = \frac{1}{S_x S_y} \sum_{s_x=0}^{S_x-1} \sum_{s_y=0}^{S_y-1} I_e\left(\mathbf{p}_{ij}^{(s_x,s_y)}\right). \tag{4}$$

Our numerical task is therefore the efficient and robust solution of the inverse-mapping problem (eq. 2) for each candidate element at the corresponding sensor point. We can use analytic inversion for linear triangles (`tri3`) and linear quadrilateral (`quad4`) elements [52], see figure 1. For higher-order elements, and optionally for `quad4`, we solve the inverse problem iteratively using Newton–Raphson as described in Section 2.3.

2.2 Camera model & transformations

We use a standard pinhole camera model in homogeneous coordinates in order to distinguish clearly between the transformed coordinates before the perspective divide used in the inverse-mapping solver and the post-divide coordinates used to address sensor points on the camera. In the following, we first describe the *forward* camera mapping from world coordinates to distorted raster space, and then the corresponding *inverse* mapping used during rasterisation. The forward camera model answers the question: given a point in the world, where does it land on the camera sensor? The inverse camera model answers the complementary question used during rasterisation: given a sensor point on the regular pixel or sub-pixel grid, from which ideal pinhole coordinate would the corresponding viewing ray have originated?

For the forward mapping, let $\mathbf{x}_w = [x_w,\ y_w,\ z_w,\ 1]^T$ denote a point in world coordinates where $\mathbf{x}_w = [\mathbf{x}_e(\boldsymbol{\xi}),\ 1]^T$, and let $\mathbf{T}_{clip} \in \mathbb{R}^{4\times4}$ denote the homogeneous camera transformation matrix mapping world coordinates into the clip-space convention used by our rasteriser. The corresponding transformed point is

$$\mathbf{x}_c = \mathbf{T}_{clip}\,\mathbf{x}_w = \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ W_c \end{bmatrix}, \tag{5}$$

where $\mathbf{x}_c$ is the clip-space coordinate, and (X_c, Y_c, Z_c, W_c) are the clip-space coordinate components before the perspective divide that are used later in the projected-residual formulation. The corresponding ideal normalised image coordinates are obtained through the perspective divide,

$$x_n = \frac{X_c}{W_c}, \qquad y_n = \frac{Y_c}{W_c}, \tag{6}$$

where $\mathbf{x}_n = (x_n, y_n)$ denotes the normalised image coordinate in the ideal pinhole camera. In the absence of lens distortion, these map to raster-space coordinates according to

$$x_r = f_x\,x_n + s\,y_n + c_x, \qquad y_r = f_y\,y_n + c_y, \tag{7}$$

where f_x and f_y are the focal lengths expressed in pixel units, (c_x, c_y) is the principal point, and s is the sensor skew parameter.

To account for lens distortion, we modify the normalised image coordinates using the Brown–Conrady distortion model. We first define r as the radial distance from the optical axis,

$$r^2 = x_n^2 + y_n^2. \tag{8}$$

The distorted normalised image coordinates (x_d, y_d) are then given by the forward distortion functions $d_x(x_n, y_n)$ and $d_y(x_n, y_n)$:

$$x_d = d_x(x_n, y_n) = x_n\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) + 2p_1 x_n y_n + p_2\left(r^2 + 2x_n^2\right), \tag{9}$$

$$y_d = d_y(x_n, y_n) = y_n\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) + p_1\left(r^2 + 2y_n^2\right) + 2p_2 x_n y_n, \tag{10}$$

where k_1, k_2, and k_3 are the radial distortion coefficients, and p_1 and p_2 are the tangential distortion coefficients. We then obtain the final distorted raster coordinates as

$$x_r = f_x\,x_d + s\,y_d + c_x, \qquad y_r = f_y\,y_d + c_y. \tag{11}$$

We may therefore summarise the complete forward mapping as

$$\mathbf{x}_w \ \rightarrow\ \mathbf{x}_c \ \rightarrow\ \mathbf{x}_n \ \rightarrow\ (x_d, y_d) \ \rightarrow\ (x_r, y_r). \tag{12}$$

Here, (x_r, y_r) denotes the raster-space location of a projected world point and is not, in general, restricted to a regular sampling grid.

For the inverse mapping used during rasterisation, we instead start from the regular sensor grid. For pixel indices (i, j) and sub-pixel indices (s_x, s_y), the corresponding sensor point is denoted by $\mathbf{p}_{ij}^{(s_x,s_y)}$ and has raster-space coordinates

$$\mathbf{p}_{ij}^{(s_x,s_y)} = \left(\bar{x}_{r,ij}^{(s_x,s_y)}, \bar{y}_{r,ij}^{(s_x,s_y)}\right), \tag{13}$$

with

$$\bar{x}_{r,ij}^{(s_x,s_y)} = i + \frac{2s_x + 1}{2S_x}, \qquad \bar{y}_{r,ij}^{(s_x,s_y)} = j + \frac{2s_y + 1}{2S_y}, \tag{14}$$

where i and j are the zero-indexed pixel coordinates and the resulting nominal raster-space coordinates are in the middle of each interval. The key point is that the regular sampling grid exists only for the sensor points $\mathbf{p}_{ij}^{(s_x,s_y)}$ in raster space. In the present work, we use the terms *clip space*, *normalised image space*, and *raster space* consistently to distinguish between the homogeneous transformed coordinates $\mathbf{x}_c$ before the perspective divide, the normalised image coordinates $\mathbf{x}_n = (x_n, y_n)$ after the perspective divide, and the floating-point raster coordinates of either a projected world point or a sensor point on the camera.

In `Riley`, the exact sensor-point evaluation remains formulated in the ideal pinhole camera, while the final output image, screen bounds, and binning operations are defined in distorted raster space. Therefore, for each sensor point $\mathbf{p}_{ij}^{(s_x,s_y)}$ with raster coordinates $(x_r, y_r) = \left(\bar{x}_{r,ij}^{(s_x,s_y)}, \bar{y}_{r,ij}^{(s_x,s_y)}\right)$, we must recover the corresponding ideal pinhole coordinate (x_n, y_n) in normalised image space. We first invert the intrinsic mapping for the target coordinate,

$$x_d = \frac{x_r - c_x - s\, y_d}{f_x}, \qquad y_d = \frac{y_r - c_y}{f_y}, \tag{15}$$

and then solve the nonlinear inverse-distortion problem

$$\mathbf{g}(\mathbf{q}) = \begin{bmatrix} g_x(x_n, y_n) \\ g_y(x_n, y_n) \end{bmatrix} = \begin{bmatrix} d_x(x_n, y_n) - x_d \\ d_y(x_n, y_n) - y_d \end{bmatrix} = \mathbf{0}, \qquad \mathbf{q} = \begin{bmatrix} x_n \\ y_n \end{bmatrix}, \tag{16}$$

where $d_x(x_n, y_n)$ and $d_y(x_n, y_n)$ are the Brown–Conrady forward distortion expressions. Starting from the initial guess $\mathbf{q}_0 = [x_d,\ y_d]^T$, we recover the ideal pinhole coordinate iteratively using Newton–Raphson,

$$\mathbf{q}_{k+1} = \mathbf{q}_k - \mathbf{J}_g(\mathbf{q}_k)^{-1}\mathbf{g}(\mathbf{q}_k), \tag{17}$$

where $\mathbf{J}_g = \partial\mathbf{g}/\partial\mathbf{q} \in \mathbb{R}^{2\times 2}$ is the Jacobian of the inverse-distortion residual. We may therefore summarise the complete inverse chain used during rasterisation as

$$(i, j, s_x, s_y) \ \rightarrow\ \mathbf{p}_{ij}^{(s_x,s_y)} \ \rightarrow\ (x_r, y_r) \ \rightarrow\ (x_d, y_d) \ \rightarrow\ \mathbf{x}_n. \tag{18}$$

The recovered normalised image coordinate $\mathbf{x}_n = (x_n, y_n)$ defines the undistorted ideal sensor point used by the inverse-mapping and shading stages of the rasterisation pipeline. The corresponding sensor points in normalised image space are generally no longer uniformly spaced after inversion of the distortion model.

2.3 Element geometry & inverse mapping

Each surface element e is defined over a domain $\hat{\Omega}_e$ with isoparametric mapping

$$\mathbf{x}_e(\boldsymbol{\xi}) = \sum_{i=0}^{n_e-1} N_i^{(e)}(\boldsymbol{\xi})\, \mathbf{x}_i, \qquad \boldsymbol{\xi} \in \hat{\Omega}_e, \tag{19}$$

where n_e is the number of nodes in element e, $N_i^{(e)}$ are the element shape functions, and $\mathbf{x}_i$ are the world coordinates of node i in element e. We use standard Lagrange shape functions and their analytic

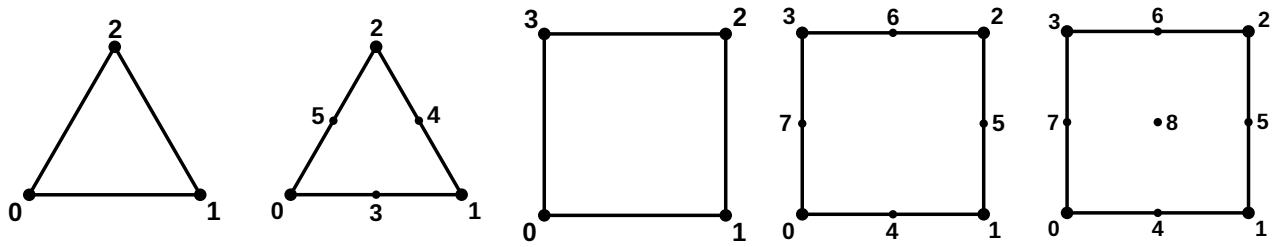
derivatives, which may be found in [40]. For quadrilateral elements, the parent coordinate is given by $\boldsymbol{\xi} = (\xi, \eta)$ with

$$-1 \leq \xi \leq 1, \qquad -1 \leq \eta \leq 1, \tag{20}$$

while for triangular elements,

$$\xi \geq 0, \qquad \eta \geq 0, \qquad \xi + \eta \leq 1. \tag{21}$$

In each case, the admissible set of parametric coordinates is denoted by $\hat{\Omega}_e$. For all element types we adopt the winding convention shown in figure 1.



(a) tri3 **(b)** tri6 **(c)** quad4 **(d)** quad8 **(e)** quad9

Fig. 1 Supported surface element types and node winding conventions used

For `tri3` elements, projection into the ideal image plane remains affine and we can therefore evaluate sensor-point inclusion directly using raster-space edge functions (eq. 22). Although edge functions are standard for linear triangle rasterisation, we introduce them here because we use the same signed-area formulation later to construct and evaluate the adaptive hulls in Section 2.4 that form part of the present higher-order rasterisation framework. Let the projected raster-space vertices of a tri3 element be $\mathbf{p}_0 = (x_{r,0}, y_{r,0})$, $\mathbf{p}_1 = (x_{r,1}, y_{r,1})$, and $\mathbf{p}_2 = (x_{r,2}, y_{r,2})$. For a directed edge from vertex a to vertex b, we define the edge function as

$$E_{ab}(x_r, y_r) = (x_r - x_{r,a})(y_{r,b} - y_{r,a}) - (y_r - y_{r,a})(x_{r,b} - x_{r,a}). \tag{22}$$

This edge-function implementation matches the winding convention and signed-area test used in our implementation. A raster-space sensor point (x_r, y_r) lies inside the projected triangle if the three edge functions $E_{01}(x_r, y_r)$, $E_{12}(x_r, y_r)$, and $E_{20}(x_r, y_r)$ have consistent sign under the chosen winding convention. The corresponding barycentric coordinates λ_i and perspective-correct interpolation of a nodal field ϕ may then be written compactly as

$$\lambda_i = \frac{E_{jk}(x_r, y_r)}{E_{jk}(x_{r,i}, y_{r,i})}, \qquad \phi(x_r, y_r) = \frac{\displaystyle\sum_{i=0}^{2} \lambda_i \, \phi_i / W_{c,i}}{\displaystyle\sum_{i=0}^{2} \lambda_i / W_{c,i}}, \qquad (i, j, k) \in \{(0,1,2), (1,2,0), (2,0,1)\}, \tag{23}$$

where $\lambda_0 + \lambda_1 + \lambda_2 = 1$, ϕ_i is the nodal value at vertex i, and $W_{c,i}$ is the corresponding clip-space perspective divisor.

For all remaining element types, projection into the image plane is nonlinear, so we perform point evaluation by solving the inverse problem in parent coordinates. Writing the clip-space coordinates of node i as $(X_{c,i}, Y_{c,i}, W_{c,i})$, the projected residual for a target ideal sensor point in normalised image space (x_n, y_n) is

$$R_x(\boldsymbol{\xi}) = \sum_{i=0}^{n_e-1} N_i^{(e)}(\boldsymbol{\xi}) \left(x_n W_{c,i} - X_{c,i} \right), \tag{24}$$

$$R_y(\boldsymbol{\xi}) = \sum_{i=0}^{n_e-1} N_i^{(e)}(\boldsymbol{\xi}) \left(y_n W_{c,i} - Y_{c,i} \right). \tag{25}$$

The inverse-mapping problem is therefore written as

$$\mathbf{R}_e(\boldsymbol{\xi}) = \begin{bmatrix} R_x(\boldsymbol{\xi}) \\ R_y(\boldsymbol{\xi}) \end{bmatrix} = \mathbf{0}, \qquad \boldsymbol{\xi} \in \hat{\Omega}_e. \tag{26}$$

This formulation avoids explicit interpolation of the already-divided image coordinates and instead solves the residual directly in clip space before the perspective divide. As a result, the effects of nonlinear projection and nonlinear element interpolation are folded into a single projected-residual equation in parent coordinates. We solve this nonlinear system using Newton–Raphson. Let $\boldsymbol{\xi}_k$ denote the parametric coordinates at iteration k. The update is then

$$\boldsymbol{\xi}_{k+1} = \boldsymbol{\xi}_k - \mathbf{J}_R(\boldsymbol{\xi}_k)^{-1}\, \mathbf{R}_e(\boldsymbol{\xi}_k)\,, \tag{27}$$

where $\mathbf{J}_R = \partial \mathbf{R}_e / \partial \boldsymbol{\xi} \in \mathbb{R}^{2\times 2}$ is the Jacobian of the projected residual,

$$\mathbf{J}_R(\boldsymbol{\xi}) = \begin{bmatrix} \displaystyle\sum_{i=0}^{n_e-1} \frac{\partial N_i^{(e)}}{\partial \xi} \left(x_n W_{c,i} - X_{c,i}\right) & \displaystyle\sum_{i=0}^{n_e-1} \frac{\partial N_i^{(e)}}{\partial \eta} \left(x_n W_{c,i} - X_{c,i}\right) \\ \displaystyle\sum_{i=0}^{n_e-1} \frac{\partial N_i^{(e)}}{\partial \xi} \left(y_n W_{c,i} - Y_{c,i}\right) & \displaystyle\sum_{i=0}^{n_e-1} \frac{\partial N_i^{(e)}}{\partial \eta} \left(y_n W_{c,i} - Y_{c,i}\right) \end{bmatrix}. \tag{28}$$

For quad4 elements, we implement both analytic inverse bilinear mapping and Newton–Raphson. We omit the analytic quad4 inverse here for brevity and because it is not central to the present higher-order framework; interested readers are referred to [52]. In practice, we use this analytic form for quad4 only in the scalar path, as it requires significant branching logic which does not lend itself well to our SIMD-over-sensor-points optimisation strategy. We retain the Newton formulation for quad4 as the preferred general formulation for SIMD evaluation over sensor points. For tri6, quad8, and quad9 elements, we use Newton–Raphson throughout. We accept a converged solution only if the residual is below tolerance, the determinant of the Jacobian is above tolerance, and the resulting parent coordinate $\boldsymbol{\xi}$ lies within the admissible parent domain $\hat{\Omega}_e$.

To the best of our knowledge, the specific use of the linearised, homogeneous projected residual defined above has not previously been applied to the direct rasterisation of higher-order finite elements. By clearing the denominator of the traditional rational projection, we transform the inverse-mapping problem into a polynomial system. This significantly reduces the complexity of the Jacobian by eliminating the need for the quotient rule and enhances numerical stability by avoiding division operations within the iterative Newton–Raphson loop, providing a robust framework for synthetic image generation from higher-order finite elements.

2.4 Adaptive hull construction & early rejection

For elements solved through the Newton–Raphson inverse mapping, we employ an adaptive hull in raster space to provide an inexpensive early-rejection test before the exact solve is attempted. The hull is constructed in two-dimensional raster coordinates, since it is used only to determine whether a raster-space sensor point lies sufficiently close to the projected element footprint to justify direct inversion. Consider one curved edge of an element in raster space, with major corner nodes $\mathbf{p}_a = (x_{r,a}, y_{r,a})$ and $\mathbf{p}_b = (x_{r,b}, y_{r,b})$, and corresponding midside node $\mathbf{p}_m = (x_{r,m}, y_{r,m})$. Using the edge function defined previously (see eq. 22), the sign of $E_{ab}(\mathbf{p}_m)$ determines on which side of the chord from $\mathbf{p}_a$ to $\mathbf{p}_b$ the midside node lies. We use this to classify the projected edge as either inward-bulging or outward-bulging with respect to the element winding convention.

If the midside node bulges inward, the hull support point associated with that edge is taken to be the projected midside node itself as illustrated in figure 2. If the midside node bulges outward, the hull support point is taken instead to be the projected quadratic Bézier control point, since the projected curved edge lies within the convex hull of its endpoints and Bézier control point. Denoting the edge support point by $\mathbf{h}_{ab}$, we then assemble the adaptive hull by walking around the element

boundary in winding order and inserting each projected major node together with the corresponding edge support point,

$$\mathcal{H}_e = \{\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{m_e-1}\} \subset \mathbb{R}^2, \tag{29}$$

where m_e is the number of points on the adaptive hull for element e.

Let $\mathbf{p}_{c,e}$ denote the projected raster-space centroid of element e. Once the ordered hull $\mathcal{H}_e$ has been constructed, we define an early-rejection region using the triangle fan

$$\mathcal{F}_e = \{\triangle(\mathbf{p}_{c,e}, \mathbf{h}_j, \mathbf{h}_{j+1}) \mid j = 0, \ldots, m_e - 1\}, \tag{30}$$

with cyclic indexing such that $\mathbf{h}_{m_e} = \mathbf{h}_0$. Using the edge-function test from the `tri3` case, we then check whether the raster-space sensor point $\mathbf{p}$ lies inside any triangle of this fan. If it does not, we reject the point without invoking the direct inverse solver. If it does, the element remains a candidate and we solve the exact inverse-mapping problem. In the present work, we use this adaptive hull procedure only for the Newton-solved element paths, since tri3 uses direct edge-function evaluation and `quad4` may additionally use analytic inversion. An example adaptive hull construction for a `tri6` element is shown in figure 2 to illustrate the procedure.

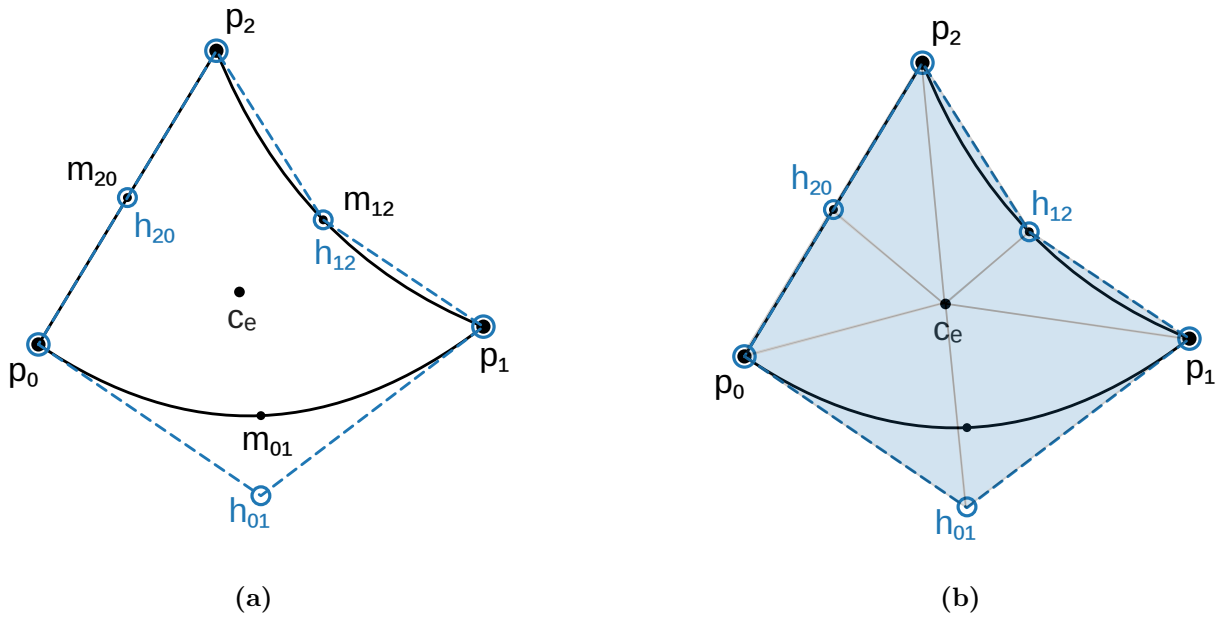


Fig. 2 Example adaptive hull construction for a `tri6` element showing (a) the ordered hull and edge support points, and (b) the resulting early-rejection triangle fan

2.5 Shader functions

The shader function s_e defines the intensity associated with a visible parent coordinate $\boldsymbol{\xi}$ on element e. In the nodal interpolation mode, each node carries an attribute $\mathbf{a}_i \in \mathbb{R}^c$, where $c = 1$ channels for greyscale rendering and $c = 3$ channels for red-green-blue (RGB) rendering. We provide support for rendering up to $c = 8$ channels, which can be useful for rendering nodal attributes; for these cases, we render a single greyscale image per channel. For all elements except `tri3`, we interpolate the nodal field directly through the element shape functions,

$$s_e(\boldsymbol{\xi}) = \sum_{i=0}^{n_e-1} N_i^{(e)}(\boldsymbol{\xi})\, \mathbf{a}_i, \tag{31}$$

where $N_i^{(e)}$ are the shape functions of element e. For tri3 elements, the corresponding nodal interpolation is evaluated using perspective-correct hyperbolic interpolation in raster space, consistent with the linear projected geometry.

In the texture-sampling mode, each i'th node carries a texture coordinate attribute $\mathbf{t}_i = [u_i,\ v_i]^T$ with $u_i, v_i \in [0, 1]$, and the interpolated texture coordinate at $\boldsymbol{\xi}$ is denoted by

$$\mathbf{t}_e(\boldsymbol{\xi}) = \begin{bmatrix} u_e(\boldsymbol{\xi}) \\ v_e(\boldsymbol{\xi}) \end{bmatrix}. \tag{32}$$

The shader function is then obtained by texture sampling a two-dimensional texture image $\mathcal{T} : [0, 1]^2 \to \mathbb{R}^c$ according to

$$s_e(\boldsymbol{\xi}) = \mathcal{T}(\mathbf{t}_e(\boldsymbol{\xi}))\,. \tag{33}$$

In the present work, we evaluate $\mathcal{T}$ using nearest, linear, cubic (Catmull-Rom, Mitchell-Netravali, bspline), Lanczos (lanczos3) or quintic (bspline) texture sampling kernels [12], with optional lookup-table accelerated forms of the higher-order kernels. We implement these modes for both greyscale and RGB textures.

In the analytic function mode, we evaluate the shader value directly from a prescribed function ,$\mathcal{F}$, rather than from stored nodal attributes or a sampled texture image. This is written as

$$s_e(\boldsymbol{\xi}) = \mathcal{F}(\boldsymbol{\xi}) \qquad \text{or} \qquad s_e(\boldsymbol{\xi}) = \mathcal{F}(\mathbf{t}_e(\boldsymbol{\xi}))\,, \tag{34}$$

depending on whether the function is defined in parent-coordinate space or texture-coordinate space. This mode is particularly useful for verification, since it permits us to construct exact analytic intensities independently of image discretisation. Rendering performance in `Riley` depends not only on the underlying algorithms, but also on software design choices that determine how effectively those algorithms are executed on modern CPU hardware. We discuss the overarching software architecture of `Riley` in the following section.

3 Computational method & software architecture

3.1 Overview of the rendering pipeline

An overview of the `Riley` rendering pipeline is shown in figure 3. The number of rendered frames is controlled by the input meshes and the number of time steps in the associated nodal attribute fields which are typically displacement fields used to deform the mesh during synthetic image generation. The external user interface of `Riley` is intentionally compact. At the top level, rendering is exposed through a single entry-point function that takes the render group specifications, camera inputs, mesh inputs, rasterisation configuration, and optional output path, and then renders the full image sequence. The corresponding Zig function signature is shown in Algorithm 1.

Algorithm 1 The external entry-point function signature for `Riley`.

```
pub fn raster(
    outer_alloc: std.mem.Allocator,
    render_groups: []const RenderGroupSpec,
    camera_inputs: []const cam.CameraInput,
    meshes: []const mo.MeshInput,
    config: RasterConfig,
    out_dir_path: ?[]const u8,
) !?ndarray.NDArray(64)
```

This single entry point reflects the intended usage model of `Riley`: the user provides the scene description, camera definitions, and render configuration, and the renderer then executes the complete multi-frame pipeline. Depending on the configuration, the rendered image stack may either be returned directly in memory or written to disk.

Moving through the internal stages of our rendering pipeline we begin by preparing our camera data from the user selected distortion model, we allocate our output frames buffer if required and then dispatch frame jobs to our task scheduler. There are two nested outer loops in our rendering

pipeline the first is over the number of frames in the scene and the second loop is applied over the number of cameras in the scene. We then process and assign work to each of the render groups input by the user. The render groups then split this work into batches of geometry work and batches or raster work. There are 4 key stages to the inner rendering pipeline: (1) coordinate transformations and mesh data preparaton for the current frame (`prepareMeshFrames(...)`); (2) tile element overlap binning (`sceneTileElemOverlap(...)`); (3) the main raster loop (`rasterScene(...)`); (4) saving the frame to disk and/or writing it to memory (`saveFrame(...)`). We separate the first two stages of the pipeline as geometry batches and the second two stages of the pipeline as raster batches which we dispatch independently as part of our hierarchical parallelisation model described in Section 3.6. The reason for this is that for typical DIC UQ work loads the geometry pre-processing front end is computationally much lighter then the raster hot loop so being able to schedule batches of geometry jobs separately increases throughput. We describe the inner workings of the key stages shown in figure 3 of rendering pipeline in the sections that follow.

3.2 Geometry pre-processing

The mesh data entering the geometry preprocessing stage is provided in a node-major format, consisting of a coordinate list of nodes and a connectivity table that indexes into this coordinate list for each mesh. From here we only operate on the data required to render the current frame. In the first geometry pre-processing stage (`prepareMeshFrames(...)`) we assume that the user has provided nodal displacement fields in a coordinate system that is consistent with the mesh, which allows us to add the displacements directly to the nodal coordinates to obtain the deformed mesh. We then take these deformed nodal coordinates and apply the homogeneous transformation into the camera reference frame described in Section 2.2.

Once the deformed coordinates for the current frame have been transformed, we apply three stages of element culling to minimise the amount of data sent to the raster hot loop. Each culling stage answers a specific question: (1) is the element in front of the camera?; (2) is the element back-facing?; and (3) does the element intersect the sensor field of view? We first check the sign of the transformed Z coordinate to determine whether an element lies behind the camera, and cull it if it does. We then perform back-face culling. For `tri3` elements, we do this using a signed-area check based on the edge function defined in eq. 22. For the higher-order element types we compute the Z component of the surface normal at all element nodes, and only cull the element if all nodal normals are back-facing. Finally, we determine whether the element can appear on the sensor. For `tri3` elements we use the axis-aligned bounding box defined by the forward-projected distorted raster coordinates from eq. 12. For the higher-order elements we instead use the axis-aligned bounding box of the adaptive hull, also formed in forward-projected distorted raster coordinates through the camera model. In other words, the cropping, culling, and tile-assignment stages are all carried out in the same distorted raster-space coordinate system as the sensor itself.

Once we have identified the elements that are both visible to the camera and not back-facing, we gather their per-element data from the node-major coordinate list and connectivity table into an element-major flat-array representation. This data reorganisation is a small but important part of the preprocessing step. The mesh and displacement fields naturally enter `Riley` in a node-major form, because that is the layout most users already have and it matches the way finite-element data is usually stored. The raster loop, however, wants the opposite view of the world. Once an element is being tested against a tile, the kernel repeatedly needs that element's nodal positions, projected coordinates, texture coordinates, and any shader attributes. We therefore gather the required node-major data into an element-major working representation before entering the hot rasterisation loop. This costs some preprocessing work, but it keeps the inner loop simple and avoids repeatedly chasing through global connectivity arrays while evaluating sensor points.

At this stage we have an element-major representation containing the data required to rasterise the visible elements per mesh for the current camera. The next task is to assign these elements to sensor tiles for processing in the raster loop in the second geometry pre-processing staged (`sceneTileElemOverlap(...)`). We do this in three passes to perform memory allocation once, outside the loops over elements and tiles. In the first pass we determine, for each tile, how many element

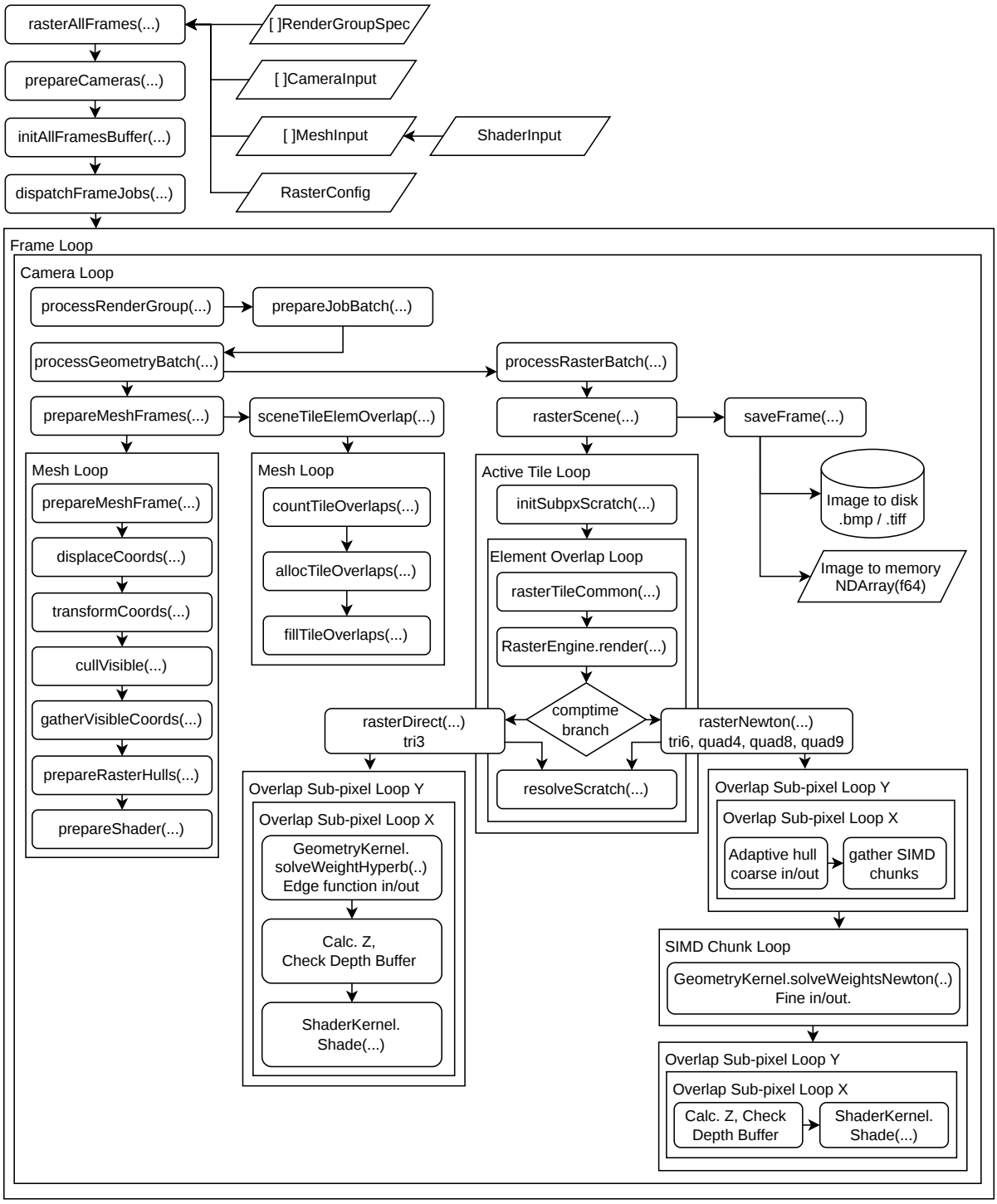


Fig. 3 Overview of the `Riley` rendering pipeline. The figure shows the main user-provided input data, the principal processing stages, the nested frame, camera, and tile loops, and the rendered image outputs. Parallelograms denote data, rounded rectangles denote processing stages, and enclosing boxes denote repeated work regions

overlaps it contains. In the second pass we perform a serial count over these per-tile overlap counts to determine both the total number of active tiles and the total number of element–tile overlaps. This gives us the information needed to allocate the memory required for these slices exactly once. The same pass also provides the prefix-sum style offsets needed to map each active tile to its range within the overlap array. In the third pass we fill these allocated slices. Each active tile stores an index into the overlap slice together with the range of overlaps it owns for processing, while each overlap record stores the corresponding index into slice of inoput meshes and the index into the element-major array for that mesh. As part of this element–tile overlap construction we store the axis-aligned bounding box of the overlap region between the element and the sensor tile. This overlap box is passed directly to the raster loop. As a result, the inner loops over sub-pixel sensor points only need to evaluate the region of the tile that can actually contain the element, rather than scanning the entire tile. This further reduces the amount of work sent to the raster hot loop.

3.3 Raster loop sensor point evaluation

The raster loop operates on the active tiles and element–tile overlaps constructed during geometry preprocessing. Each active tile owns a contiguous range of overlap records, and each overlap record identifies both the element to be tested and the axis-aligned overlap box between the element footprint and the tile. This means that the raster loop does not scan the whole image, or even necessarily the whole tile, for each candidate element. Instead, it only evaluates the sub-pixel sensor points inside the overlap region that was already identified during tile construction.

Before entering the sub-pixel loops for an element, we hoist the required element data from the element-major arrays into small local arrays. These local arrays hold the nodal coordinates, projected coordinates, shape-function data, texture coordinates, and any shader attributes needed by the selected geometry and shader kernels. This is a simple but useful step: we access the element data repeatedly while looping over the sub-pixel sensor points in the overlap region, so we want it to be local, compact, and hot in cache before the inner loop begins.

We handle visibility using a tile-local depth buffer stored at the sub-pixel level. Each sub-pixel sensor point has an associated depth value, and candidate element contributions are only accepted if they are closer to the camera than the value already stored in the buffer. This lets us resolve visibility before shading where possible. The depth buffer is local to the tile because each tile is processed independently and writes to a unique region of the output image. This avoids synchronisation in the raster hot loop while still providing per-sub-pixel visibility.

The raster loop has two main sensor-point evaluation paths. The first is the direct path used for `tri3` elements. Since a linear triangle projects to a triangle in raster space, we can use the edge-function test directly to determine whether a sub-pixel sensor point lies inside the element footprint. The resulting direct evaluation procedure is summarised in Algorithm 2. If the point passes the in/out test and the depth-buffer test, we compute the barycentric weights and use the perspective-correct hyperbolic interpolation pathway described earlier. The point is then sent directly to the shader kernel.

The second path is used for higher-order elements, and optionally for lower-order elements when the Newton pathway is selected. In this case, the projected element footprint is not generally a simple straight-sided polygon, so we first apply the adaptive hull check as a coarse in/out test. This early-rejection test is summarised in Algorithm 3. Sensor points that pass this check are gathered into small batches for Newton iteration. The Newton solve maps the regular raster-space sensor point back to the parent coordinates of the candidate element using the projected residual formulation. The full Newton-path point evaluation procedure is summarised in Algorithm 4. We use the centroid of the element as the initial guess for the Newton solver. We then shade points that converge, satisfy the residual tolerance, pass the admissible parent-domain test, and pass the depth-buffer test, and discard points that fail any of these checks. In this way, failures in the inverse solve are treated as ordinary rejection cases rather than exceptional conditions.

This structure is deliberately conservative in where expensive work is placed. Cheap tests are performed first: the overlap box limits the sensor-point loop bounds, the adaptive hull removes points that are clearly outside a curved element, and the depth buffer avoids unnecessary shading for points

that are already occluded. The more expensive Newton solve and shader evaluation are only applied to points that have survived these earlier stages.

Algorithm 2 Direct `tri3` sensor-point evaluation

Require: raster-space sensor point $\mathbf{p}$, precomputed ideal pinhole coordinate $\mathbf{x}_n(\mathbf{p})$, `tri3` element data, tile-local depth buffer, shader kernel
Ensure: reject point or accumulate shaded contribution
Compute edge-function values for $\mathbf{p}$ using the projected triangle vertices
if $\mathbf{p}$ is outside the triangle within the in/out tolerance **then**
    reject point
end if
Compute perspective-correct barycentric weights using $\mathbf{x}_n(\mathbf{p})$
Compute depth $z(\mathbf{p})$ using the perspective-correct `tri3` pathway
if $z(\mathbf{p})$ fails the tile-local depth-buffer test **then**
    reject point
end if
Update the tile-local depth buffer at $\mathbf{p}$
Evaluate the shader kernel using the perspective-correct barycentric weights
Accumulate the shaded value into the sub-pixel scratch buffer

Algorithm 3 Adaptive hull early-rejection test

Require: raster-space sensor point $\mathbf{p}$, projected element centroid $\mathbf{p}_{c,e}$, ordered adaptive hull points $\{\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{m_e-1}\}$
Ensure: reject element early or continue to Newton-path evaluation
Set `inside_hull` to `false`
for each adaptive hull edge $j = 0, \ldots, m_e - 1$ **do**
    Let $\mathbf{h}_j$ and $\mathbf{h}_{j+1}$ be the two hull points on the current edge, with the index wrapped around at the end of the hull
    Form the fan triangle with vertices $\mathbf{p}_{c,e}$, $\mathbf{h}_j$, and $\mathbf{h}_{j+1}$
    Test whether $\mathbf{p}$ lies inside this fan triangle using the `tri3` edge-function test from eq. 22
    if $\mathbf{p}$ is inside the current fan triangle **then**
        Set `inside_hull` to `true`
        Stop testing the remaining fan triangles
    end if
end for
if `inside_hull` is `true` **then**
    Continue to Newton-path evaluation
else
    Reject the element for this sensor point
end if

3.4 Shading, texture sampling, & resolve

Once a sub-pixel sensor point has passed the geometric, inverse-mapping, and visibility checks, we send it to the shader kernel to evaluate the image intensity at that location. The shader stage is intentionally separated from the geometry kernel. The geometry kernel is responsible for deciding whether a sensor point belongs to an element and for providing the interpolation weights or parent coordinates required by the shader. The shader kernel then uses this information to compute the actual greyscale or RGB value that will be accumulated into the sub-pixel scratch buffer.

The simplest shader pathway is nodal interpolation. In this mode, the user provides nodal attributes and `Riley` interpolates those attributes to the accepted sensor point. For `tri3` elements we use the perspective-correct hyperbolic interpolation weights computed during the direct edge-function

Algorithm 4 Newton-path sensor-point evaluation

Require: raster-space sensor point $\mathbf{p}$, precomputed ideal pinhole coordinate $\mathbf{x}_n(\mathbf{p})$, higher-order element data, adaptive hull, tile-local depth buffer, shader kernel

Ensure: reject point or accumulate shaded contribution

Apply the adaptive hull early-rejection test from Algorithm 3
if $\mathbf{p}$ fails the adaptive hull test **then**
    reject point
end if
Initialise the parent coordinate estimate $\boldsymbol{\xi}_0$
Solve for the parent coordinates $\boldsymbol{\xi}$ using Newton–Raphson on the projected residual with target coordinate $\mathbf{x}_n(\mathbf{p})$
if Newton iteration does not converge **then**
    reject point
end if
if the final residual is above tolerance **then**
    reject point
end if
if $\boldsymbol{\xi}$ lies outside the admissible parent domain **then**
    reject point
end if
Compute depth $z(\mathbf{p})$ from the converged element position
if $z(\mathbf{p})$ fails the tile-local depth-buffer test **then**
    reject point
end if
Update the tile-local depth buffer at $\mathbf{p}$
Evaluate the shader kernel using $\boldsymbol{\xi}$ and the element shape functions
Accumulate the shaded value into the sub-pixel scratch buffer

pathway in Algorithm 2. For all other supported element types we use the converged parent coordinates from the inverse solve in Algorithm 4 and evaluate the corresponding finite-element shape functions, as described in eq. 31. This shader is useful when the image intensity, displacement, or any other scalar or vector field is naturally attached to the mesh nodes.

The texture shader follows the same first step, but applies it to nodal texture coordinates. Each element carries nodal coordinates in texture space, usually denoted by (u, v), and these are interpolated to the accepted sensor point using the same geometric weights or parent coordinates used by the nodal shader. The resulting texture coordinate is then used to reconstruct the image intensity from the input texture. This is the main pathway used for rendering deformed speckle pattern images for DIC uncertainty quantification. We support both greyscale and RGB textures, with a dedicated greyscale path because this is the common case for DIC and avoids unnecessary channel work.

We support several texture reconstruction kernels. These include nearest-neighbour and linear sampling, cubic kernels such as B-spline, Catmull–Rom, and Mitchell–Netravali, a Lanczos-3 kernel, and a quintic B-spline kernel. For the higher-order reconstruction kernels we provide three evaluation modes: direct function evaluation, lookup-table evaluation, and lookup-table evaluation with linear interpolation between table entries. We write the direct formulations using Horner-style polynomial evaluation where appropriate. For the lookup-table modes, the table size is a compile-time constant and the table itself is generated at compile time, with a default size of 1024 entries. This allows the user to trade a small amount of memory for reduced reconstruction cost in texture-heavy workloads.

The third shader pathway is the analytic function shader. This mode is mainly included for verification and controlled numerical experiments, rather than as the default route for DIC image generation. Analytic functions can be evaluated either directly from the element parent coordinates and weights, or from interpolated user-provided texture coordinates. The supported functions include constant, linear, quadratic, sinusoidal, smoothed checkerboard, and a simple Lambertian normal-based intensity. The function parameters are user controlled, which makes this path useful for constructing known reference images where interpolation, projection, anti-aliasing, and visibility behaviour can be tested independently of a sampled input texture.

All shader pathways write into a tile-local sub-pixel scratch buffer rather than directly into the final resolved image. This is because `Riley` supports sub-sample anti-aliasing by evaluating multiple sensor points per output pixel. Once all relevant element contributions for a tile have been processed, the scratch buffer is resolved by averaging the sub-pixel values associated with each pixel as per eq. 4. We provide separate dedicated greyscale and RGB resolve paths so that the common greyscale case used in DIC does not pay for unnecessary channel handling. The resolved pixel values are then written into the output image buffer for the tile, after which we can either save the frame to disk or copy it into the returned in-memory image stack.

3.5 Kernel specialisation & SIMD execution

We compose our raster engine with compile-time specialised kernels rather than a single generic runtime-dispatched kernel. This specialisation spans all supported combinations of geometry kernel (i.e element type as `tri3`, `tri6`, etc.) and shader kernel (nodal interpolated, texture and analytic function). Each kernel type carries a set of compile-time known constants together with a small collection of inline functions implementing the relevant evaluation pathway. By resolving the element-type and shader-type combinations at compile time, rather than repeatedly dispatching over them inside the raster loop, we allow the compiler to emit specialised code paths for each rendering case. The associated compile-time constants then enable constant propagation, dead-branch elimination, and simplification of control flow inside the hot loop. This provides a useful middle ground between a monolithic generic kernel with runtime dispatch and a fully hand-written set of separate raster loops for every case. In this way, the mathematical pathways remain explicit in the source code while most dispatch overhead is shifted from runtime to compile time. This strategy also makes the raster loop more SIMD-friendly by reducing branch-heavy logic in the vectorised path. The main tradeoff we make here is that compilation times are increased for improved run time performance.

We targeted our explicit SIMD optimisations at the raster-loop stage of the rendering pipeline, as preliminary single-threaded performance experiments showed this stage to be the dominant compu-

tational cost for representative DIC UQ workloads. Our implementation uses explicit SIMD vector instructions targeting a 512-bit width, corresponding to $S = 8$ SIMD lanes of 64-bit floating-point values (`f64` in Zig). For nodal interpolation shaders, we retain this outer SIMD-over-sensor-points strategy throughout the full evaluation path. In particular, we execute the higher-order Newton pathway as a three-stage SIMD-batched process in order to keep the computation S-wide over sensor points for as long as possible. First, we perform the adaptive hull check S-wide and compact the passing points into a pre-allocated buffer. Second, we process this buffer in S-wide chunks through the Newton solve and compact the converged points that pass the inverse-mapping stage into a second pre-allocated buffer. Third, we invoke the shader on the surviving points. For nodal interpolation shaders, this final stage also remains S-wide over sensor points. For texture shaders, however, we found that a different SIMD strategy was preferable. There, we switch to an inner-SIMD approach in which SIMD is applied within a single texture sample evaluation rather than across multiple samples. This proved especially advantageous for the cubic and quintic reconstruction kernels, where performance is dominated less by arithmetic throughput than by the memory traffic associated with fetching the required texel neighbourhood (4x4 texels for cubic and 6x6 for quintic texels). In these higher-order texture kernels, inner-SIMD therefore provided a better match to the computational structure of the interpolation than the outer SIMD-over-sensor-points strategy used for the front end of the raster loop.

3.6 Parallel execution strategy

We have implemented a hierarchical parallel execution model in `Riley` which separates coarse-grained scheduling over frame–camera jobs from fine-grained scheduling within each image for geometry preprocessing and the raster loop. At the outer level, the user provides one or more `render_groups`. Each render group has an associated Zig `io` object and a user-specified worker budget. In the current implementation, each render group is driven by a dedicated `std.Thread`, while the work internal to that render group is scheduled onto the render group's own `io` worker pool. In this way, we preserve two levels of parallelism: outer parallelism over independent frame–camera jobs assigned to different render groups, and inner parallelism within each render group for geometry preprocessing and rasterisation.

We support both in-order and offline rendering. For in-order rendering, output images are guaranteed to be produced in chronological order. In this case, we advance one time step at a time and distribute the camera jobs for that time step across the available render groups. This preserves the nested frame-loop then camera-loop execution order while still allowing concurrency across cameras and within each image. For offline rendering, frame arrival order is not guaranteed. Here, the combined frame–camera job space is treated as a single pool of work items and render groups dynamically claim batches of these jobs. This throughput-oriented mode allows multiple time steps and multiple cameras to be processed concurrently.

Parallel resource control is now expressed per render group rather than only through a single global thread count. The user specifies the number of render groups, the workers assigned to each group, the frame-batch size per group, the maximum number of geometry jobs that may be active concurrently within a group, the maximum workers allowed per geometry job, and the maximum workers allowed for the single active raster job within that group. In this way, we allow the user to guide resource usage based on the target workload. For example, a representative DIC UQ workload for quasi-static testing might process 10–100 images for a stereo pair of 24 MPx cameras, where it is often beneficial to allocate relatively more workers to rasterisation. By contrast, simulation of a high-speed camera for a dynamic test will typically involve either a single camera or a stereo pair with much lower resolution (on the order of 1 MPx), but the user may require 1000 or more output images. In that case, the outer render-group parallelism and the number of frame–camera jobs in flight become more important.

Within each render group, we use Zig's `io` task interface together with a lightweight parallel chunk executioner to express parallel `for`-style loops. This executioner maps a contiguous range of work items onto a bounded number of worker tasks without exposing low-level scheduling details throughout the raster codebase. We support both static chunk execution and dynamic chunk execution with an atomic work counter. The latter is used when load balancing is more important than minimising scheduling

overhead.

The geometry stage is itself hierarchical. A render group may process one or more geometry jobs in a wave, where each geometry job corresponds to one prepared frame–camera job. We support two explicit scheduling policies. In *spread* mode, the render-group workers are first spread across as many geometry jobs as possible, assigning one worker per job before increasing the worker count of any individual job. In *pack* mode, workers are packed into a single geometry job before additional geometry jobs are started. An *auto* mode selects between these policies at runtime from the total scene size. This reflects the fact that geometry preprocessing is often relatively regular and, for many representative workloads, is best executed single-threaded or with only modest intra-job parallelism. When additional parallelism is beneficial, we exploit both inner parallelism within a geometry job and outer parallelism by preprocessing multiple frame–camera jobs concurrently across render groups.

We then reuse the same chunk-execution machinery in the raster stage, where it is applied to tile processing. At this stage, each render group executes at most one raster job at a time, but that job may use multiple workers. The raster loop operates over the subset of active tiles identified during the geometry stage, and these tiles are processed using dynamically assigned work grains. This gives us a consistent fine-grained parallel execution model across both geometry preprocessing and rasterisation while still allowing the scheduling policy to reflect the different workload characteristics of the two stages.

Our tile-based execution model is well suited to thread-level parallelism because each tile maps to a unique region of the output image buffer. By ensuring that each worker writes only to isolated image regions, we avoid write contention in the main raster loop and therefore do not require atomics or mutex-based synchronisation for image buffer writes. We do, however, use lightweight atomic operations and bounded synchronisation in the schedulers for dynamic task assignment, geometry-job coordination, and render-group work distribution. This keeps synchronisation out of the hot rendering path and limits it to coarse-grained work management. In this way, we make a deliberate trade-off: rather than eliminating scheduling overhead entirely, we retain a small amount of inexpensive synchronisation in the scheduler in order to achieve better load balancing when geometry jobs or tile costs vary.

Finally, for disk output we optionally support overlapping image saving with rendering. In this mode, each render group maintains a small bounded pool of persistent frame buffers together with a separate single-worker save path. Once a frame has completed rasterisation, ownership of its buffer can be handed to the save path while the render group continues to the next frame–camera job. This mode is used only for disk saving and leaves the in-memory output path unchanged. It is primarily intended to reduce idle time when the disk save cost would otherwise stall a render group.

3.7 Build & portability

We have implemented `Riley` as a dependency-free software rasteriser in Zig. Aside from the Zig compiler itself, the codebase does not rely on external numerical, graphics, or runtime libraries for its core rendering and mathematical kernels. This keeps the build process simple and improves reproducibility, as the software compiles out of the box with a standard Zig toolchain. The implementation used in the present work targets Zig compiler version 0.16.0 [53]. This choice also supports portability across platforms, since Zig provides a self-contained build system and cross-compilation support within the compiler toolchain. In practice, this means that `Riley` can be built and deployed with minimal platform-specific configuration, which is advantageous for both reproducible research workflows and future open-source adoption.

4 Verification

The accuracy of a synthetic image generator for DIC uncertainty quantification cannot be established solely by applying DIC to its rendered output. While such a comparison is useful for assessing end-to-end consistency, it does not cleanly isolate rendering errors from the systematic errors embedded in the DIC measurement chain or correctness of the DIC engine itself. For this reason, we have focused our verification analysis on renderer specific tests that are independent of the downstream DIC. For

our verification analysis we have used four test cases: (1) Newton solver verification and reprojection error; (2) silhouette tests on disorted single elements; (3) pixel sub-sampling refinement convergence; and (4) visibility tests for the culling stages and overlapping elements.

Across all verification cases we used a constant camera pixel size of 5.3×10^{-6} m, a constant focal length of 50×10^{-3} m, and the pinhole camera model with no lens distortion. For all cases pixel sub-sampling was set to $SS = 1$ apart from verification case 3 where this is intentionally varied. Finally, for verification cases 2 to 4 where we are using the full rendering pipeline, we use an analytic function shader (eq. 34) to render a constant value. This constant analytic shader allows for direct comparisons of the rendered output image to ground truth for these cases.

4.1 Verification case 1: solver reprojection error

The purpose of this verification case is to isolate correctness of the Newton solver for the projected residual in eq. 26 independent of the main rendering pipeline. We assess this using the reprojection error of the solver. This is constructed by starting with a dense grid of $\boldsymbol{\xi}_{\text{true}}$ samples in the valid domain $\hat{\Omega}_e$ for a single element. This grid of parametric coordinates is then forward projected into world coordinates with a specific implementation pathway that is independent of the main codebase. We then apply our projected residual Newton solver to recover the original grid of $\boldsymbol{\xi}_{\text{rec}}$ coordinates before finally forward projecting the recovered coordinate to raster-space. We define the reprojection error ϵ_{rp} as the Euclidean distance in the raster-space sensor plane between the original projection and the projection of the recovered coordinate:

$$\epsilon_{\text{rp}} = \|\mathbf{p}_{\text{rec}} - \mathbf{p}_{\text{true}}\|, \tag{35}$$

where $\mathbf{p}_{\text{true}} = (x_r, y_r)$ is the raster-space sensor point obtained by projecting the ground-truth parent coordinate $\boldsymbol{\xi}_{\text{true}}$, and $\mathbf{p}_{\text{rec}}$ is the corresponding point for the recovered coordinate $\boldsymbol{\xi}_{\text{rec}}$. For this verification case, we use an ideal pinhole camera model, allowing the mapping chain to be simplified to:

$$\boldsymbol{\xi}_{\text{true}} \xrightarrow{\Pi} \mathbf{p}_{\text{true}} \xrightarrow{\mathcal{K}^{-1}} \mathbf{x}_n \xrightarrow{\text{Solve } \mathbf{R}_e=\mathbf{0}} \boldsymbol{\xi}_{\text{rec}} \xrightarrow{\Pi} \mathbf{p}_{\text{rec}}, \tag{36}$$

where Π denotes the forward pinhole projection to raster space, $\mathcal{K}^{-1}$ is the inverse of the linear intrinsic camera mapping, and $\mathbf{x}_n$ is the normalized image coordinate used directly by the Newton solver for the projected residual $\mathbf{R}_e$.

As we are focused on verification of the Newton solver in isolation we analyse all elements except `tri3`. We note that the `tri3` element is handled by a direct edge function and perspective correct hyperbolic interpolation solver which we will verify in the second verification case. We chose two single element distortion cases to analyse the reprojection error: the first is an affine shear distortion and the second is a midside node bulge from outwards to inward. The `quad4` element does not have midside nodes so for this case we only analyse the shear distortion. For both distortion cases the reference element is equilateral with edge length L. We set the maximum shear distortion magnitude to be L and for the midside node bulge cases we set the radial distance the midside node is displaced away from or towards the centroid to be $0.3L$ relative to the equilateral element case. We note here that we also analysed additional rotation, stretch, and tangential midside-node sweeps during development and these can be found in the test suites of the codebase, but the cases reported here were chosen because they exercise the distinct numerical behaviours of the implemented pathways: the regular reference case, an affine distortion case, and a curved-boundary case for the higher-order elements.

We define the following cases where we observe our solver is stable and all points in the parent domain $\boldsymbol{\xi}$ were recovered. For `quad4` elements, these stable cases are: (1) regular equilateral element with no distortion; and (2) shear distortion equal to L. For `tri6` elements the observed stable cases are: (1) regular equilateral element with no distortion; (2) shear distortion of L; (3) outward midside node bulge of $0.25L$; and (4) inward midside node bulge of $0.1L$. For `quad8` and `quad9` elements the observed stable cases are: (1) regular equilateral element with no distortion; (2) shear distortion of L; (3) outward midside node bulge of $0.2L$; and (4) inward midside node bulge of $0.15L$. This shows that the limiting factor for the inward bulge is the determinant of the Jacobian approaching zero and the limiting factor for an outward bulge is self-intersection. Based on the geometry of triangular elements

we expect the stable range to be more robust to outward bulging and less robust to inward bulging compared to quadrilateral elements which aligns with our results.

The results for the stable cases in table 1 and figures 4 and 5 produced pixel-space reprojection errors far below the image discretisation scale. Regular and affine-shear geometries typically gave errors below 10^{-12} px, while the more challenging midside-node bulge cases remained below approximately 10^{-10} px. Since the Newton solve uses a projected-residual tolerance of 10^{-8} and a Jacobian determinant rejection tolerance of 10^{-12}, these results show that the accepted solutions are not merely admissible, but reproject to the target sensor points to essentially numerical precision. The larger errors observed for the bulge cases are expected because these geometries approach the limits of the element mapping, either through near-zero Jacobian determinants for inward bulging or self-intersection for outward bulging. The iteration counts follow the same pattern. For the regular and affine-shear cases, the solver typically converged within two Newton iterations. For the more challenging bulge cases, this increased to around five or six iterations. This is expected because the bulged geometries introduce stronger nonlinearity into the parent-to-image mapping and, near the stability limits, move the element closer to either a near-singular Jacobian for inward bulging or self-intersection for outward bulging. Even in these more difficult cases, the solver still converged in a small number of iterations and the resulting reprojection errors remained many orders of magnitude smaller than one pixel.

In figure 5 we plot maps of the solver reprojection error for the bulge tests, including both the stable cases. Here, the limit cases correspond to bulge magnitudes beyond the stable cases reported previously. Solver non-convergence for otherwise valid parametric coordinates is clearly visible in figure 5 for the inward-bulging `tri6` case, where the corners of the element begin to become unstable. Similar behaviour is observed for the inward-bulging `quad8` and `quad9` elements, although the missing bands of valid parametric coordinates are more subtle. This is consistent with the expected failure mode for inward bulging, where the element mapping approaches a near-zero Jacobian determinant. For the stable cases, the full parametric domain converges and the reprojection errors remain very small, with errors generally below 10^{-10} px even for the more challenging bulge geometries. The largest errors are concentrated near the most distorted parts of the element boundary, which is also expected because these regions have the strongest local curvature and are closest to the stability limit of the mapping. For outward bulging, the limiting behaviour is instead associated with self-intersection or folding of the projected element boundary. In these cases the solver may still converge locally, but the mapping is no longer globally admissible for rendering because multiple parent-coordinate locations can correspond to the same projected sensor region.

Taken together, these maps show both sides of the solver behaviour. Inside the admissible stable regime, the projected-residual Newton pathway recovers the inverse mapping to essentially numerical precision over the full parent domain. Once the element geometry is pushed beyond this regime, the error maps show the expected onset of non-convergence or non-admissible mappings. We therefore treat these results as verification that the projected-residual Newton pathway is accurate and robust within the admissible element geometries used by the rasterisation pipeline.

4.2 Verification case 2: silhouette tests on distorted elements

In this verification case we test the full rendering pipeline, from input data through to the output image, using distorted single-element silhouette tests. The purpose is different from Verification Case 1: here we are no longer isolating only the Newton solve, but instead checking that the complete pipeline produces the correct rendered element footprint. The element centroid location, projected area, and binary silhouette mask are used as reference quantities. We use the same two element-distortion families as in Verification Case 1, namely affine shear and midside-node bulging.

For these tests we used the camera parameters outlined in Section 2.2 and a 1024×1024 pixel sensor with one sensor point per pixel. We used a constant analytic function shader so that the rendered image was a binary silhouette with a known intensity. For both distortion cases, the elements were rotated by 20° from the sensor horizontal so that their edges were not aligned with the pixel grid. This reduces, although does not remove, pixel-discretisation effects and helps isolate the behaviour of the rendering pipeline itself. In all cases, we constructed the element distortion so that the continuous

Table 1: Newton inverse solver reprojection accuracy for stable element geometries in verification case 1.

Element	Case	RMS ϵ_{rp} [px]	Max ϵ_{rp} [px]	Mean iters.
`quad4`	regular	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	2.000
`quad4`	shear	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	2.000
`tri6`	regular	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	2.000
`tri6`	shear	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	2.000
`tri6`	bulge inward	8.31×10^{-12}	5.78×10^{-11}	5.411
`tri6`	bulge outward	7.59×10^{-12}	5.80×10^{-11}	5.684
`quad8`	regular	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	2.000
`quad8`	shear	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	2.000
`quad8`	bulge inward	5.18×10^{-12}	4.01×10^{-11}	5.392
`quad8`	bulge outward	5.26×10^{-12}	3.96×10^{-11}	5.373
`quad9`	regular	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	2.000
`quad9`	shear	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	2.000
`quad9`	bulge inward	5.18×10^{-12}	4.01×10^{-11}	5.392
`quad9`	bulge outward	5.26×10^{-12}	3.96×10^{-11}	5.373

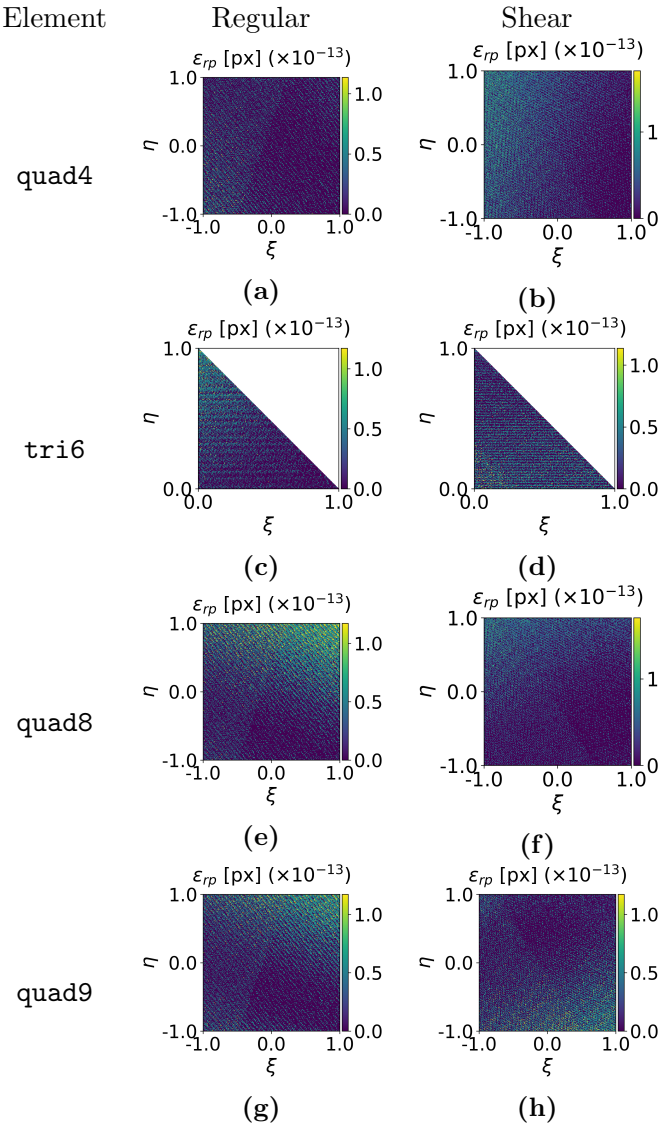


Fig. 4 Newton inverse solver reprojection error maps for the shear test in verification case 1.

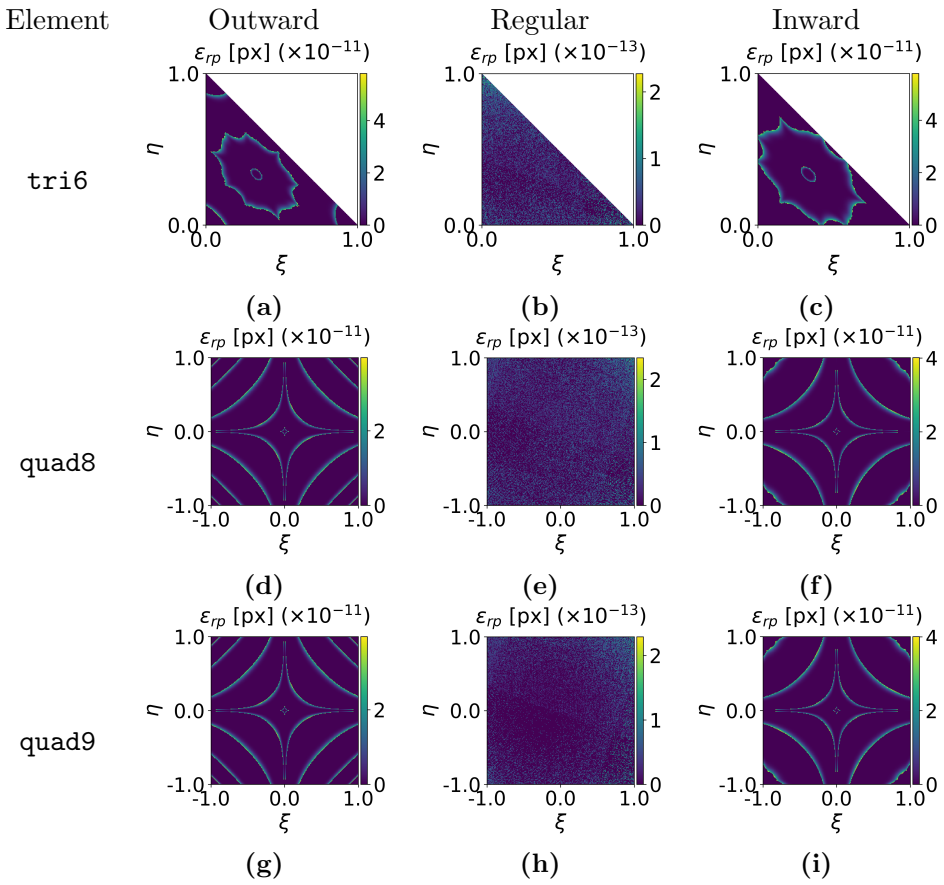


Fig. 5 Newton inverse solver reprojection error maps for the bulge tests in verification case 1.

element centroid remained at the centre of the sensor.

We generated the reference data independently of the rendering pipeline. We placed each element in a two-dimensional plane parallel to the sensor, with its centroid centred on the camera. We positioned the camera automatically so that the largest distorted element in the sweep remained inside the field of view. Using the ideal pinhole camera model, we then calculated the pixel-per-length scaling factor and used this to determine the expected nodal locations on the sensor in pixel units. From these nodal locations, we computed the projected element area using Green's theorem to convert the area integral into a boundary line integral. For the higher-order elements, the contribution of each quadratic edge was evaluated using two-point Gaussian quadrature, giving the expected area in pixel units. To construct the reference mask, we used the same nodal locations to build a fine linear discretisation of the element boundary and classified each pixel centre using a winding-number point-in-polygon test.

Using this reference data, we report the x, y, and radial r centroid errors in pixels, the percentage area error, and the percentage reference-mask error. The affine shear results are given in table 2 for all supported element types, while the midside-node bulge results are given in table 3 for the higher-order elements. For the affine shear cases, the rendered masks agree exactly with the independently generated reference masks for all element types. The area errors are also negligible, with the largest value being approximately 10^{-3}%. The quadrilateral cases show centroid errors at numerical precision, while the triangular cases show small non-zero centroid offsets of order 10^{-3} px to 10^{-2} px.

We interpret these small triangular centroid offsets as pixel-discretisation effects rather than rendering errors. The key point is that the reference-mask error is zero, meaning that the renderer and the independent reference classify the same pixels as belonging to the silhouette. The centroid error is instead measured relative to the continuous target centroid. For triangular elements, it is difficult for all three edges to intersect the pixel grid at similarly shallow angles, even after rotating the element by 20°. As a result, the stair-step errors along the discrete boundary do not cancel as cleanly as they do for the quadrilateral cases, where opposite edges provide a more balanced discretisation. This explains why the triangle cases show small centroid offsets even though the mask and area agreement

are essentially exact.

The midside-node bulge cases show the same overall behaviour. The rendered masks agree exactly with the independently generated reference masks with area errors no more than $3 \times 10^{-3}\%$. The largest centroid error occurs for the inward-bulging `tri6` case, where the radial centroid error is approximately 3.7×10^{-2} px. This is still far below the pixel scale and is consistent with the stronger boundary curvature and reduced cancellation of discretisation error for the triangular geometry. The `quad8` and `quad9` cases again show centroid errors at numerical precision, with very small area errors.

Representative rendered silhouettes for the affine shear cases are shown in figure 6, and the midside-node bulge cases are shown in figure 7. The stable cases render cleanly for all supported element types, with no visible artefacts in the element interiors or along the expected silhouettes. Near the inward-bulge stability limit, visible artefacts begin to appear, consistent with the solver behaviour observed in Verification Case 1 where the element mapping approaches a near-zero Jacobian determinant. The outward-bulge limit cases are less visually obvious in a binary silhouette because self-overlap remains filled by the same constant intensity. In these cases, the solver reprojection maps from Verification Case 1 are more informative than the rendered silhouette alone because they expose the loss of global admissibility in the element mapping.

Overall, this verification case shows that the full rendering pipeline reproduces independently generated single-element silhouettes for all supported element types. For the stable affine and bulge cases, the rendered masks match the reference masks exactly, area errors no more than $3 \times 10^{-3}\%$, and centroid errors remain far below one pixel. Rendering artefacts only appear near the expected singular or non-admissible limits of the underlying quadratic mappings, such as extreme inward bulging or outward self-intersection. These regimes lie outside the practically relevant range for the intended DIC uncertainty-quantification workflow, where the finite-element input is expected to remain geometrically sensible.

Table 2: Silhouette test result for affine deformation using pixel sub-sampling $SS = 1$ for verification case 2.

Element	Case	Centroid Err. x [px]	Centroid Err. y [px]	Centroid Err. r [px]	Area Err. [%]	Ref. Mask Err. [%]
`tri3`	regular	1.73×10^{-3}	-5.15×10^{-3}	5.43×10^{-3}	0.000	0.000
`tri3`	shear	1.91×10^{-3}	1.67×10^{-3}	2.53×10^{-3}	0.000	0.000
`quad4`	regular	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.000	0.000
`quad4`	shear	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.001	0.000
`tri6`	regular	1.73×10^{-3}	-5.15×10^{-3}	5.43×10^{-3}	0.000	0.000
`tri6`	shear	1.91×10^{-3}	1.67×10^{-3}	2.53×10^{-3}	0.000	0.000
`quad8`	regular	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.000	0.000
`quad8`	shear	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.001	0.000
`quad9`	regular	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.000	0.000
`quad9`	shear	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.001	0.000

4.3 Verification case 3: sub-sample refinement study

For this verification case we analyse silhouette convergence with increasing pixel sub-sampling. This can be thought of as the rasterisation equivalent of a mesh-refinement study in finite-element simulations: as the number of sensor points per pixel is increased, the resolved image should approach a stable reference image. In this verification case and the next, we use a multi-element mesh based on the silhouette of a rabbit to test the renderer under more realistic conditions than the single-element cases above. To generate these meshes, we discretised a vector-graphics outline of a rabbit and meshed it with all supported element types using Gmsh [54].

We swept the following pixel sub-sampling levels: $SS = 1, 2, 4, 8, 16, 32$, where SS denotes the number of sub-samples per pixel direction. The finest case, $SS = 32$, was used as the reference image. For each coarser sub-sampling level, we calculated the RMSE in 8-bit grey levels relative to

Table 3: Silhouette verification for the midside node bulge deformation using pixel sub-sampling $SS = 1$ for verification case 2.

Element	Case	Centroid Err. x [px]	Centroid Err. y [px]	Centroid Err. r [px]	Area Err. [%]	Ref. Mask Err. [%]
`tri6`	inward bulge	3.23×10^{-2}	1.85×10^{-2}	3.72×10^{-2}	0.001	0.000
`tri6`	regular	5.28×10^{-3}	-2.49×10^{-3}	5.83×10^{-3}	0.000	0.000
`tri6`	outward bulge	-8.06×10^{-3}	1.55×10^{-2}	1.74×10^{-2}	0.001	0.000
`quad8`	inward bulge	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.003	0.000
`quad8`	regular	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.000	0.000
`quad8`	outward bulge	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.002	0.000
`quad9`	inward bulge	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.003	0.000
`quad9`	regular	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.000	0.000
`quad9`	outward bulge	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	$< 1.00 \times 10^{-12}$	0.002	0.000

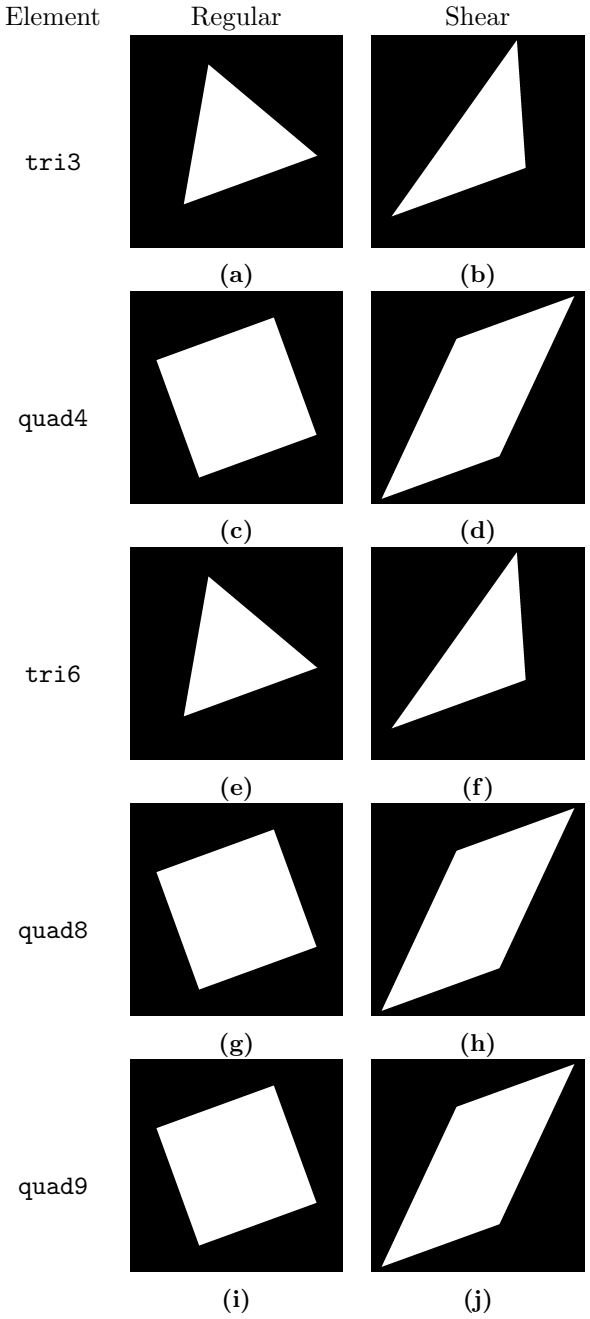


Fig. 6 Silhouette verification images for the affine shear cases in verification case 2.

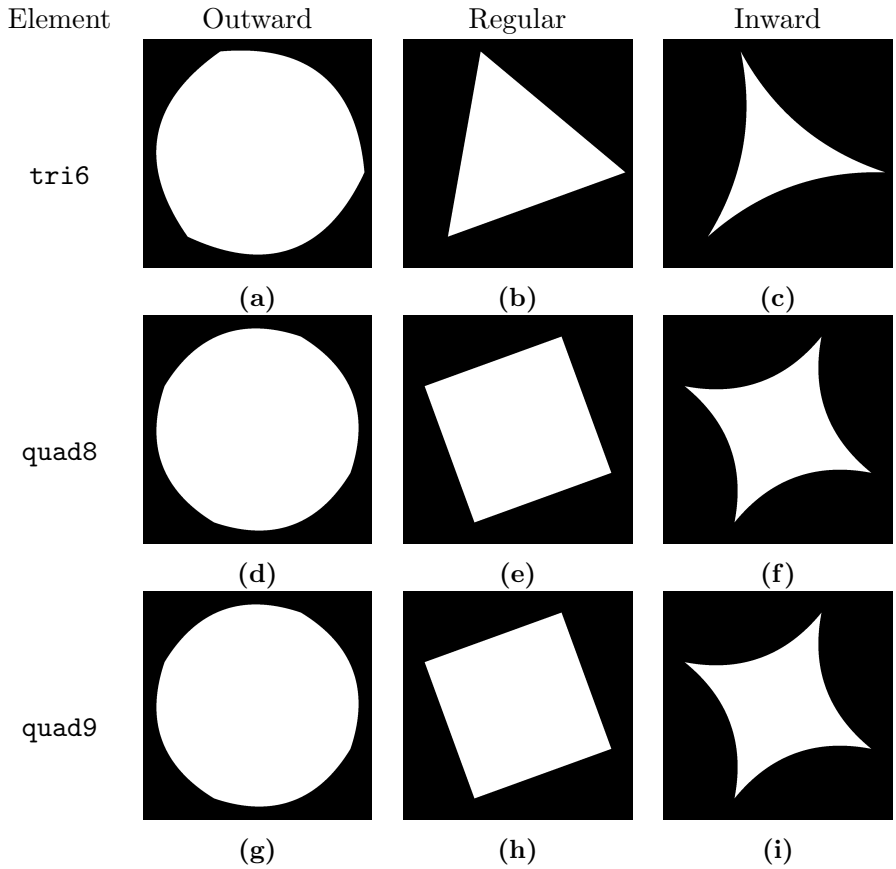


Fig. 7 Silhouette verification images for the nonlinear bulge cases in verification case 2.

this reference. We performed the analysis at two image resolutions. The first was a low-resolution image of 160×100 pixels, where the effect of pixel sub-sampling is easy to see visually. The second was a higher-resolution image of 640×400 pixels, which gives a more representative view of how the error behaves as the pixel size is reduced.

In figure 8 we show the reference render, $SS = 32$, for the `tri6` rabbit mesh, together with the coarsest render, $SS = 1$. We also show heat maps of the absolute difference between each lower sub-sampling level and the $SS = 32$ reference. The rendered silhouettes show the expected stair-step pixelisation pattern for $SS = 1$, while the reference image has a much smoother blended outline. The difference maps show that the error is concentrated almost entirely along the boundary of the rabbit silhouette, with the largest differences occurring for the $SS = 1$ case. This is the behaviour we want to see: increasing sub-sampling changes the fractional coverage of boundary pixels, rather than changing the bulk intensity of the object interior.

In figure 9 we plot the RMSE in 8-bit grey levels against sub-sampling level for the low-resolution render in figure 9(a) and for the higher-resolution render in figure 9(b). For both image resolutions, the RMSE decreases rapidly and monotonically as the sub-sampling level is increased. The largest improvement occurs when moving away from one sensor point per pixel, after which the error reduction becomes progressively smaller. Although the curves appear close to exponential when plotted against sub-sampling level, the horizontal axis is logarithmic in SS. We therefore interpret the result more conservatively as rapid, power-law-like convergence towards the high-sub-sampling reference.

The higher-resolution case gives lower RMSE values than the low-resolution case. This is expected because the boundary region occupies a smaller fraction of the total image and each pixel covers a smaller region of the projected geometry. The convergence curves therefore confirm the intended behaviour of the sub-sampling resolve: the image is already converged in the silhouette interior, and increasing SS progressively refines the boundary pixels where partial coverage matters. This verifies that the sub-sampling implementation behaves consistently for all supported element types under a more realistic multi-element rendering case.

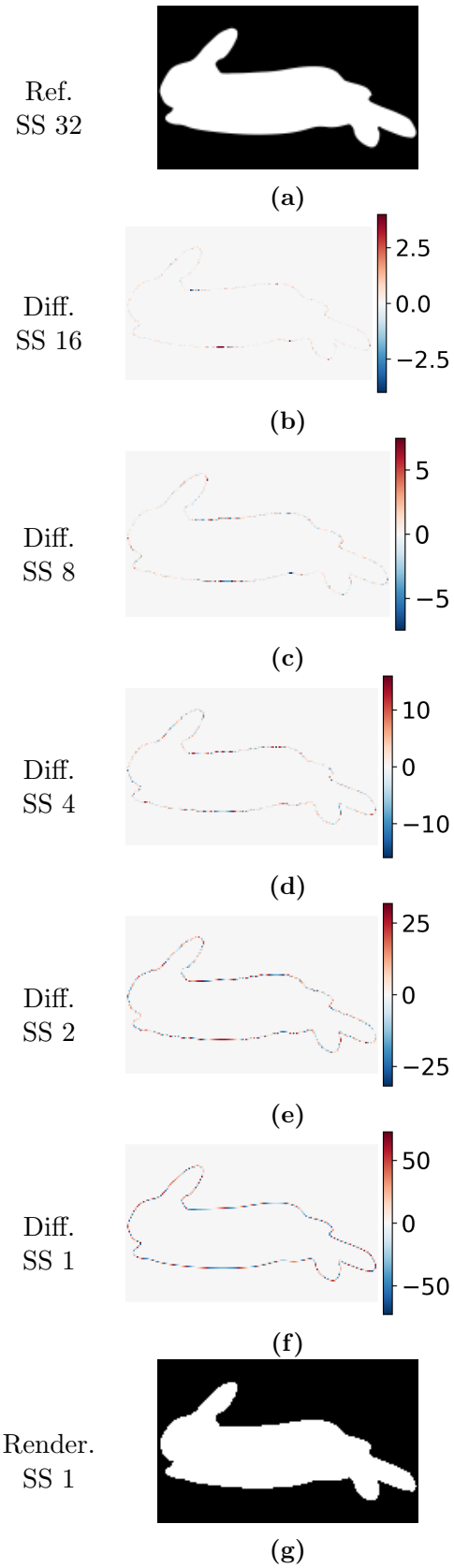


Fig. 8 Anti-aliasing convergence for the `tri6` rabbit mesh with a constant analytic function shader at low resolution for verification case 3. *SS* denotes pixel sub-sampling. The reference render is shown at the top, followed by the difference maps in descending *SS* order, and the lowest-SS render at the bottom. The difference images are plotted in 8-bit grey levels denoted by GL.

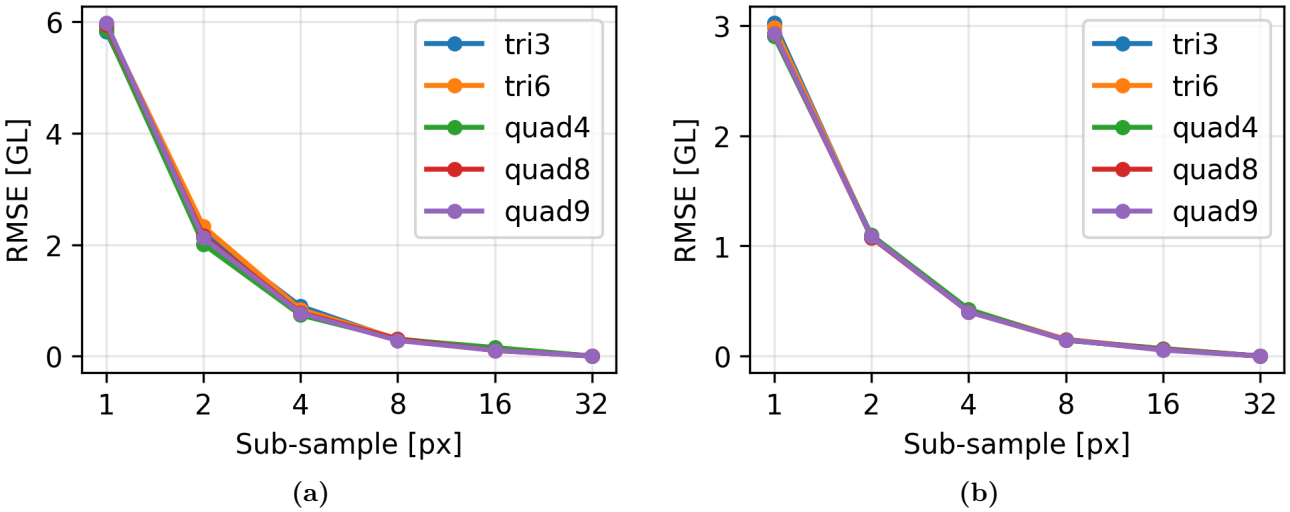


Fig. 9 Convergence of the root mean square error (RMSE) for the low-resolution (a) and high-resolution (b) difference images for verification case 3. *SS* denotes pixel sub-sample and GL denotes 8-bit grey levels.

4.4 Verification case 4: visibility tests

The purpose of this test case is to verify that the depth buffer correctly resolves visibility when two meshes overlap in the rendered image. In particular, the mesh closest to the camera should overwrite the rear mesh wherever both meshes project to the same sensor location, without any blending in the interior of the front object. To test this behaviour we again used the rabbit meshes from the previous verification case. Two rabbit meshes were placed in the scene with a prescribed horizontal and vertical overlap, with one rabbit positioned behind the other. We used constant analytic function shaders so that the expected image values were unambiguous: the front rabbit was rendered at the full dynamic range, the rear rabbit at half the dynamic range, and the background at zero intensity. With this setup, any leakage of the rear mesh through the front mesh would appear immediately as grey contamination within the white front-rabbit silhouette.

To also exercise mixed-mesh rendering, we tested cases where the front and rear rabbits used different supported element orders while keeping the element shape family consistent. For example, when the front rabbit used `tri6` elements, the rear rabbit used `tri3` elements. Similarly, quadrilateral cases paired different orders within the supported quadrilateral element family. This means that the test checks not only the depth-buffer update itself, but also that visibility is resolved correctly when multiple meshes with different geometry kernels are present in the same scene.

We performed a simple numerical check by rendering a second image using the same camera setup but with only the front white rabbit present. We then counted the number of pixels at full dynamic range in the front-only image and compared this with the number of full-dynamic-range pixels in the two-rabbit image. If the depth buffer is functioning correctly, adding the rear rabbit should not change the set of pixels assigned to the front rabbit, so these two counts should be identical. This is what we observed for all tested element combinations.

An example render from this verification case is shown in figure 10. The two-rabbit image shows the rear grey rabbit only where it is not overlapping the front rabbit, while the front rabbit remains uniformly white over its visible interior. The difference image between the two-rabbit render and the front-only render further isolates the visible part of the rear rabbit and shows no residual contribution within the front-rabbit silhouette. This verifies that the tile-local depth buffer correctly prevents rear-mesh contributions from overwriting or blending with closer visible geometry.

(a) **(b)** **(c)**

Fig. 10 Visibility test images for the `tri6` rabbit meshes in verification case 4: (a) both meshes, (b) front mesh only, and (c) the difference image.

5 Performance

5.1 Benchmark setup

We analysed four benchmark cases selected to measure the main performance characteristics of `Riley`. These benchmarks target the dominant parts of the rendering pipeline, the effect of the key implementation optimisations, and thread scaling for a representative DIC UQ workload. The four cases are: (1) raster-loop throughputl; (2) geometry-preprocessing throughput; (3) adaptive-hull and SIMD ablation; and (4) thread scaling on a representative DIC UQ workload.

For the first three benchmarks we used a common single-camera setup with a 1600×1000 pixel sensor, pixel sub-sampling of $SS = 1$, a pixel size of 5.3×10^{-6} m, and a focal length of 50×10^{-3} m. We ran these benchmarks single-threaded so that the measured timings isolate the cost of the relevant pipeline stage and implementation pathway. For the first three benchmarks we returned the rendered image in memory and specifically excluded disk input/output (io) overhead from the benchmarks. The fourth benchmark analyses thread scaling on a larger representative DIC UQ case, and the details of that setup are described separately in Section 5.5.

For each benchmark configuration we performed 25 timed runs. We report wall-clock timings using the median and median absolute deviation (MAD), and also record the minimum and maximum values for diagnostic checks. We embedded timing instrumentation directly in the benchmark executable using `std.Io.Clock` from the Zig standard library. We ran all benchmarks on a desktop workstation with an AMD Ryzen Threadripper 7980X processor (64 cores), 256 GB of DDR5 RAM, and a 1 TB NVMe SSD, running Ubuntu 24.04.2 LTS. We compiled the benchmarks with Zig 0.16.0 [53] using the `-O ReleaseFast` optimisation flag.

5.2 Benchmark 1: raster loop performance

For this benchmark we analysed raster-loop throughput using the minimum number of elements required to fill the whole sensor. This setup ensures that all pixels pass the visibility checks and are sent to the shader for evaluation. For triangular elements, `tri3` and `tri6`, this requires two elements joined along the image diagonal. For quadrilateral elements, `quad4`, `quad8`, and `quad9`, a single element is sufficient to cover the sensor.

We instrumented `Riley` to capture the wall-clock time from entry into the raster engine loop until exit from the loop, denoted t_{raster}. This excludes geometry preprocessing, which is analysed in the next benchmark, as well as setup and input/output time. We then calculated raster throughput in mega-pixels per second (MPx/s) as $1.6/t_{\text{raster}}$, since the sensor contains 1.6 MPx and this benchmark uses $SS = 1$.

We investigated raster performance for all supported element types using three shaders: (1) nodal interpolation; (2) Catmull–Rom cubic texture sampling with direct evaluation using Horner's method; and (3) Catmull–Rom cubic texture sampling using a 1024-entry lookup table (LUT) with linear interpolation between table entries. The texture was a representative DIC speckle pattern mapped linearly to the element, with the interpolated uv coordinates lying in the range 0.4–0.6. The nodal interpolation shader gives a useful estimate of the raw throughput of the shading pathway without texture reconstruction overhead, while the cubic texture shaders provide a more realistic estimate of

throughput for synthetic speckle-image generation.

All results in this benchmark use the production raster kernels with both SIMD execution and adaptive hulls enabled. This is intentionally the same pathway used in normal rendering, but it is worth noting that the full-sensor fill case is not especially favourable to the adaptive-hull optimisation. For quadrilateral elements, the element bounding box and the sensor overlap almost exactly, so there is little opportunity for hull-based rejection. For the triangular cases, the two triangles fill the rectangular sensor together, but each triangle still carries a rectangular bounding box that includes points outside the triangular footprint. These points are rejected by the adaptive hull before reaching the inverse solve, which adds some work in this artificial full-screen benchmark. The adaptive-hull contribution is therefore analysed separately in the optimisation ablation in Section 5.4.

The raster-loop throughput results are shown in table 4. We report the end-to-end time, the raster-loop time, and the raster-loop throughput for each case. The difference between the end-to-end and raster timings is approximately constant across the table, at about 3.6×10^{-3} s. This confirms that the variation between cases is dominated by the raster loop itself, rather than fixed setup overhead.

As expected, the `tri3` nodal interpolation case is the fastest configuration, reaching 242 MPx/s on a single thread. This is the direct non-iterative pathway and does not require the Newton solve used by the higher-order and `quad4` kernels. Among the Newton-solved cases, `quad4` gives the highest nodal-interpolation throughput at approximately 86 MPx/s, followed by the higher-order elements. The `tri6`, `quad8`, and `quad9` nodal cases all fall in the range of roughly 56–60 MPx/s.

The `tri6` case is slightly disadvantaged in this benchmark because of the way the sensor-space bounding boxes are constructed. The two triangular elements together cover the rectangular sensor, but each individual triangle has a bounding box that covers half of the sensor rectangle. This means that many sensor points inside the bounding box are rejected at the adaptive-hull stage before they reach the Newton solve. The quadrilateral cases do not have this additional rejection work in this benchmark because the element bounding box and the sensor overlap almost exactly. This makes the quadrilateral cases a cleaner measure of the Newton and shader cost, while the triangular cases also include the cost of rejecting points outside the triangular footprint.

The cubic texture shaders are substantially more expensive than nodal interpolation, which is also expected. The texture shader first interpolates the nodal uv coordinates, which is essentially the same interpolation task performed by the nodal shader, and then uses those coordinates to reconstruct the texture intensity. For the Newton-solved elements, the Catmull–Rom LUT-lerp texture shader achieves roughly half the throughput of the corresponding nodal interpolation shader. For the direct `tri3` path, where the nodal shader is especially fast, the texture shader achieves about one fifth of the nodal-interpolation throughput. The LUT-lerp texture kernel achieves very similar performance to direct Catmull–Rom evaluation using Horner's method, with direct evaluation being slightly faster by approximately 1–2% across all element types. This indicates that the LUT-lerp implementation provides comparable performance to direct evaluation without the need for additional memory lookup, although both offer alternative pathways for DIC speckle rendering.

Overall, these results show that the single-threaded production raster loop is already fast for both direct and Newton-solved element pathways. The direct `tri3` case provides a useful upper bound on the cost of the simplest raster path, while the higher-order texture cases represent a more demanding and realistic workload for DIC UQ image synthesis. These timings also provide the baseline for the optimisation ablation in Section 5.4, where we analyse how adaptive hulls and SIMD execution further improve raster-stage throughput under a more representative higher-order workload.

5.3 Benchmark 2: geometry throughput

For this benchmark we used approximately 1×10^5 elements of each type to mesh a rectangular plate filling the whole sensor. We instrumented `Riley` to capture the wall-clock time from entry into the geometry-preprocessing pipeline through to exit, denoted t_{geometry}. This timing includes the addition of displacements to deform the mesh, coordinate-system transformations, coarse visibility and element-culling checks, packing of the remaining visible elements, and element–tile binning and data reshaping ready for the raster loop. We calculated the throughput of the geometry pipeline in mega-nodes and mega-elements processed per second (MNodes/s and MElem/s) using $N_{\text{geom}}/t_{\text{geometry}}$, where N_{geom}

Table 4: Raster stage performance results for the full sensor rendering cases in benchmark 1. Timings and throughputs are reported as median ± median absolute deviation (MAD).

Element Type	Shader	Element Count	End-to-end [10^{-3} s]	Raster [10^{-3} s]	Throughput [MPx/s]
`tri3`	nodal interp.	2	10.25 ± 0.051	6.62 ± 0.031	241.57 ± 1.152
`tri3`	Catmull–Rom direct	2	34.12 ± 0.050	30.51 ± 0.043	52.45 ± 0.074
`tri3`	Catmull–Rom LUT-lerp	2	34.67 ± 0.048	31.05 ± 0.044	51.53 ± 0.072
`tri6`	nodal interp.	2	32.07 ± 0.067	28.41 ± 0.077	56.32 ± 0.153
`tri6`	Catmull–Rom direct	2	51.95 ± 0.096	48.13 ± 0.099	33.24 ± 0.069
`tri6`	Catmull–Rom LUT-lerp	2	52.72 ± 0.137	48.90 ± 0.124	32.72 ± 0.083
`quad4`	nodal interp.	1	22.36 ± 0.078	18.71 ± 0.125	85.52 ± 0.567
`quad4`	Catmull–Rom direct	1	42.28 ± 0.111	38.63 ± 0.090	41.42 ± 0.097
`quad4`	Catmull–Rom LUT-lerp	1	43.10 ± 0.055	39.46 ± 0.046	40.55 ± 0.047
`quad8`	nodal interp.	1	30.39 ± 0.123	26.74 ± 0.118	59.84 ± 0.264
`quad8`	Catmull–Rom direct	1	51.13 ± 0.167	47.49 ± 0.143	33.69 ± 0.102
`quad8`	Catmull–Rom LUT-lerp	1	51.87 ± 0.131	48.22 ± 0.133	33.18 ± 0.092
`quad9`	nodal interp.	1	31.31 ± 0.104	27.67 ± 0.108	57.83 ± 0.225
`quad9`	Catmull–Rom direct	1	51.87 ± 0.191	48.22 ± 0.175	33.18 ± 0.121
`quad9`	Catmull–Rom LUT-lerp	1	52.54 ± 0.067	48.88 ± 0.070	32.73 ± 0.047

is either the exact number of input nodes or the exact number of input elements for the case. We report both throughput measures because the first part of the geometry pipeline operates naturally on node-major data, while the later stages are driven by element-major data.

The geometry benchmark results are shown in table 5. For this deliberately large mesh case, the geometry-preprocessing time accounts for approximately 34–56% of the end-to-end time, depending on element type. Most cases sit close to 38–40%, with `tri3` being higher because the raster stage for the direct pathway is fast. This is a useful stress test for the geometry front end: the mesh is much denser than we would normally expect for a DIC UQ rendering problem, and the element size is approximately a pixel or less.

The nodal throughput gives a useful view of the node-major part of the pipeline. In these results, `quad9` gives the highest nodal throughput at approximately 9.0 MNodes/s, followed by `quad8` and `tri6` at about 7.1–7.3 MNodes/s. This is consistent with the data-access pattern in the geometry preprocessor: higher-order elements fetch more nodal data per connectivity entry, so some of the per-element connectivity and loop overhead is amortised over more nodes. By contrast, `tri3` has the lowest nodal throughput because each element contributes relatively few nodes, so the per-element overhead is paid more often for each node processed.

The element-wise throughput shows the opposite trend, which is also expected. The `tri3` case has the highest element throughput at approximately 9.5 MElem/s. This pathway does not require adaptive hull construction and uses a cheap signed-area check for back-face culling, so the per-element preprocessing work is low. The `quad4` case is the next fastest at approximately 5.2 MElem/s. Although it uses the Newton pathway in the raster loop, its geometry preprocessing remains relatively simple because it has only four nodes and no curved edges. The higher-order elements are slower in element-throughput terms because each element requires more nodal data, more projected quantities, and additional work to construct the curved-element footprint used for tile assignment. This gives throughputs of approximately 3.6 MElem/s for `tri6`, 2.4 MElem/s for `quad8`, and 2.3 MElem/s for `quad9`.

This split between nodal and element throughput is important. A high-order element can look efficient when measured per node, because more nodal work is grouped into each element traversal. The same case can look slower when measured per element, because the element itself carries more data and requires more preprocessing. Reporting both quantities therefore gives a more honest picture of the geometry pipeline than either metric alone.

We note that 1×10^5 elements is significantly larger than what we expect for many practical DIC UQ studies, where meshes containing 1×10^3 to 1×10^4 surface elements are more common. In that more typical range, geometry preprocessing should contribute much less to the end-to-end runtime than it does in this deliberately dense benchmark. The raster loop scales primarily with the number of sensor points and the shader/inverse-mapping cost, while the geometry stage scales with the number and order of the surface elements. This benchmark therefore represents a geometry-heavy case rather than the expected balance for most image-synthesis workloads.

Overall, the geometry-preprocessing pipeline is fast enough that it is unlikely to be the dominant cost for the intended DIC UQ use cases. Even for this dense plate mesh, the full geometry stage is processed in approximately 20–56 $\times 10^{-3}$ s on a single thread, depending on element type. For the more common case where the mesh is one to two orders of magnitude smaller, the geometry cost should be a small part of the total render time. This supports the design choice made throughout `Riley`: spend a modest amount of preprocessing work to reshape finite-element data into a form that makes the raster loop simpler and faster.

5.4 Benchmark 3: optimisation ablation study

In this benchmark we analyse the effect of two key optimisations: adaptive hulls for early rejection of sensor points before the Newton solve, and explicit SIMD execution in the raster loop. We focus on the element types that use the Newton pathway, which includes `quad4`, `tri6`, `quad8`, and `quad9`. The `tri3` element is excluded from this benchmark because it uses the direct edge-function pathway and therefore does not exercise the same optimisation structure. We note here that we have implemented the same raster loop SIMD optimisations for `tri3` and these are reflected in the raster loop throughput

Table 5: Geometry-preprocessing benchmark results for representative element types. The image size is fixed at 1600×1000 pixels, SSAA is fixed at 1×1, and all runs are single threaded. Timings and throughputs are reported as median $\pm$ MAD over 10 runs. Wall-clock timings are reported in 10^{-3} seconds. The geometry throughput is reported in both MNodes/s and MElem/s, computed from the exact input node and element counts and the geometry preprocessing time.

Element Type	Node Count	Element Count	End-to-end [10^{-3} s]	Geometry [10^{-3} s]	Throughput [MNodes/s]	Throughput [MElem/s]
`tri3`	103041	204800	38.59 ± 0.045	21.50 ± 0.028	4.79 ± 0.006	9.53 ± 0.013
`tri6`	410881	204800	148.20 ± 0.262	56.48 ± 0.148	7.27 ± 0.019	3.63 ± 0.010
`quad4`	103041	102400	58.57 ± 0.127	19.71 ± 0.031	5.23 ± 0.008	5.20 ± 0.008
`quad8`	308481	102400	113.04 ± 0.386	43.31 ± 0.308	7.12 ± 0.051	2.36 ± 0.017
`quad9`	410881	102400	113.59 ± 0.194	45.49 ± 0.087	9.03 ± 0.017	2.25 ± 0.004

for Benchmark 1 (Section 5.2).

For this benchmark we used a spherical mesh containing 2000 elements of each type. The camera was placed automatically so that the sphere filled as much of the field of view as possible. This gives a more representative rendering case than the full-sensor single-element benchmark above, because the elements are distributed across the image and appear at a range of orientations relative to the camera. We used the Catmull–Rom cubic texture shader with LUT-lerp evaluation, using the same representative DIC speckle texture as in the previous benchmarks. This gives a demanding but realistic raster workload for synthetic DIC image generation.

The adaptive-hull optimisation is tightly integrated into both the geometry-preprocessing stage and the raster loop. During geometry preprocessing, the adaptive hulls are used to form tight sensor-space bounding boxes for coarse visibility checks and tile–element overlap construction. During rasterisation, the same hulls are then used as an early-rejection test before the Newton solve. Building a fair comparison case therefore required more than simply switching off the raster-loop hull test. We also needed a replacement front-end bounding strategy that remained robust for curved higher-order element edges.

For this comparison, we implemented a padded bounding-box baseline, denoted padded-BB. This baseline uses the minimum and maximum projected nodal locations to form a sensor-space bounding box, with an additional safety factor to account for higher-order element edges that can bulge beyond the nodal limits. We selected the padding by regression testing against a series of midside-node bulge cases rendered with the adaptive-hull path. This led to a padding factor of 15% of the projected nodal span. We then disabled the adaptive-hull early-rejection test in the raster loop, so that all points passing the padded-BB were sent directly to the Newton solver. This gives a complete padded-BB comparison path for the adaptive-hull implementation.

The SIMD comparison is more direct. The first version of `Riley` was written as a scalar implementation, and we retained this pathway as a reference while developing the SIMD kernels. We use compile-time selected pathways to build either the scalar or SIMD version of the raster pipeline. This lets us compare the same rendering workload with and without explicit SIMD execution. The SIMD implementation used in this benchmark is applied to the raster loop only, since initial experiments showed little benefit from SIMD in the geometry-preprocessing stage.

We evaluated the full ablation matrix for each Newton-solved element type. The reference configuration is the scalar padded-BB case. table 6 reports the wall-clock times for the end-to-end, geometry-preprocessing, and raster stages, along with raster throughput. table 7 reports the corresponding speedup factors relative to the scalar padded-BB baseline.

The geometry timings show the expected trade-off: adaptive hull construction is slightly more expensive than the padded-BB front end. For example, in the SIMD configurations, the geometry time increases from approximately 4.48×10^{-3} s to 4.62×10^{-3} s for `tri6`, from 4.06×10^{-3} s to 4.20×10^{-3} s for `quad8`, and from 4.10×10^{-3} s to 4.22×10^{-3} s for `quad9`. This is the cost of constructing a tighter and more useful raster-space footprint. The important result is that this small preprocessing cost is more than recovered in the raster loop.

The speedup results show that both optimisations are effective, and that their combined effect is strongest for the higher-order elements. For `tri6`, SIMD alone gives a raster speedup of 4.68×, adaptive hulls alone give 5.83×, and the combined adaptive-hull SIMD path gives 12.99×. For `quad8` and `quad9`, the combined raster speedups are similarly strong, reaching 8.92× and 8.72×, respectively. In end-to-end terms, the combined speedups are 12.15× for `tri6`, 8.19× for `quad8`, and 8.02× for `quad9`. These are substantial gains for a single-threaded benchmark.

The combined speedups are larger than a purely additive improvement for the higher-order elements, but they are not multiplicative. This is expected because the adaptive hulls and SIMD kernels are tightly coupled optimisations rather than independent changes. The hulls reduce the number of sensor points that reach the expensive inverse-mapping path, while SIMD accelerates the remaining point evaluations. Each optimisation therefore changes the amount and structure of the work available for the other optimisation to accelerate.

The `quad4` case benefits least from the combined optimisation, although it still achieves a raster speedup of 3.56× and an end-to-end speedup of 3.24×. This is also expected. The `quad4` element has no midside nodes and no curved element edges, so the adaptive hull has less work to remove compared with the higher-order cases. The Newton solve for `quad4` also converges in fewer iterations, so skipping points before the Newton solve has a smaller payoff.

Overall, this ablation study shows that the two main raster optimisations in `Riley` are highly effective for the intended higher-order rendering workloads. Adaptive hulls reduce unnecessary Newton solves, and SIMD execution accelerates the remaining sensor-point evaluations. When combined, these optimisations produce approximately 8–12× end-to-end speedups for the higher-order elements in this single-threaded benchmark, and up to 13× speedup in the raster loop itself. This gives a strong foundation for the thread-scaling study in the next section, where the already-optimised tile-based raster loop is parallelised across workers.

Table 6: Optimisation ablation timings for adaptive hulls and SIMD execution in benchmark case 3. Timings and throughputs are reported as median ± median absolute deviation (MAD).

Element	Bounds	Kernel	End-to-end [10^{-3} s]	Geometry [10^{-3} s]	Raster [10^{-3} s]	Raster [MPx/s]
`quad4`	padded BB	scalar	104.72 ± 0.095	3.86 ± 0.034	100.69 ± 0.074	15.89 ± 0.012
`quad4`	padded BB	SIMD	51.48 ± 0.098	3.81 ± 0.009	47.52 ± 0.104	33.67 ± 0.074
`quad4`	adaptive hull	scalar	48.42 ± 0.065	3.85 ± 0.030	44.43 ± 0.038	36.01 ± 0.031
`quad4`	adaptive hull	SIMD	32.28 ± 0.071	3.83 ± 0.012	28.30 ± 0.062	56.53 ± 0.124
`tri6`	padded BB	scalar	875.77 ± 0.286	4.63 ± 0.012	870.60 ± 0.297	1.84 ± 0.001
`tri6`	padded BB	SIMD	191.01 ± 0.224	4.48 ± 0.007	186.07 ± 0.227	8.60 ± 0.010
`tri6`	adaptive hull	scalar	154.42 ± 0.137	4.69 ± 0.047	149.26 ± 0.113	10.72 ± 0.008
`tri6`	adaptive hull	SIMD	72.10 ± 0.106	4.62 ± 0.007	67.01 ± 0.102	23.88 ± 0.037
`quad8`	padded BB	scalar	396.27 ± 0.144	4.08 ± 0.029	391.91 ± 0.140	4.08 ± 0.001
`quad8`	padded BB	SIMD	103.57 ± 0.104	4.06 ± 0.005	99.23 ± 0.101	16.12 ± 0.016
`quad8`	adaptive hull	scalar	107.79 ± 0.101	4.23 ± 0.040	103.27 ± 0.067	15.49 ± 0.010
`quad8`	adaptive hull	SIMD	48.39 ± 0.050	4.20 ± 0.010	43.93 ± 0.067	36.42 ± 0.055
`quad9`	padded BB	scalar	405.18 ± 0.142	4.12 ± 0.018	400.71 ± 0.133	3.99 ± 0.001
`quad9`	padded BB	SIMD	108.58 ± 0.070	4.10 ± 0.013	104.16 ± 0.070	15.36 ± 0.010
`quad9`	adaptive hull	scalar	108.02 ± 0.115	4.31 ± 0.047	103.45 ± 0.077	15.47 ± 0.012
`quad9`	adaptive hull	SIMD	50.51 ± 0.059	4.22 ± 0.014	45.98 ± 0.051	34.80 ± 0.039

5.5 Benchmark 4: representative thread scaling for DIC UQ

For the realistic DIC UQ threading benchmark we analysed a linear elastic thin plate with a hole subjected to tensile loading. We generated a mesh of `quad20` elements using Gmsh [54] and solved the model using the MOOSE solid mechanics module [55, 56]. We then used tools in the `pyvale` Python package to extract the surface mesh and associated nodal displacement data as `quad8` elements to be

Table 7: Optimisation speedup matrix for adaptive hulls and SIMD execution in benchmark case 3. Speedup multipliers are computed relative to the padded-BB scalar baseline for each element type. Values greater than one indicate faster execution.

Element	Configuration	End-to-end speedup	Raster speedup
`quad4`	SIMD only	2.03	2.12
`quad4`	adaptive hull only	2.16	2.27
`quad4`	adaptive hull + SIMD	3.24	3.56
`tri6`	SIMD only	4.58	4.68
`tri6`	adaptive hull only	5.67	5.83
`tri6`	adaptive hull + SIMD	12.15	12.99
`quad8`	SIMD only	3.83	3.95
`quad8`	adaptive hull only	3.68	3.79
`quad8`	adaptive hull + SIMD	8.19	8.92
`quad9`	SIMD only	3.73	3.85
`quad9`	adaptive hull only	3.75	3.87
`quad9`	adaptive hull + SIMD	8.02	8.72

passed to `Riley`. The bottom edge of the plate was fixed and a vertical displacement was applied to the top edge, with the maximum displacement increasing linearly to 0.01 mm over the 64 time steps in the simulation. Additional model input parameters and the associated Gmsh and MOOSE input scripts are provided in the `Riley` Git repository [51].

The rendering setup used two cameras, each with a 2464×2056 pixel sensor, corresponding to approximately 5 MPx per image. The pixel size was 3.45 μm and the focal length was 50 mm. One camera was placed face-on to the plate and the second camera was placed at a stereo angle of 20°. We used pixel sub-sampling of $SS = 2$, so each rendered image required four sensor-point evaluations per output pixel before resolve. Images were either returned directly in memory or written to disk in 8-bit BMP format. For this benchmark we used the Catmull–Rom cubic texture shader with direct evaluation, as this is the representative production texture pathway for synthetic speckle-image generation. The input speckle texture is included in the `Riley` repository. Texture coordinates were generated using a simple planar projection onto the flat surface of the specimen, which is sufficient for this benchmark because the plate remains a planar surface while deforming in-plane. This gives a realistic DIC rendering workload: a stereo camera pair, multi-frame FE displacement data, higher-order surface elements, texture reconstruction, pixel sub-sampling, and optional disk output.

All runs in this benchmark used the offline rendering mode, allowing frames to be rendered and saved asynchronously. We also used the geometry `spread` scheduling mode. In this mode, each render group processes its geometry jobs in batches using a single worker per batch before moving on to raster jobs. This keeps the geometry preprocessing simple and avoids over-parallelising a stage that is small compared with the raster work for this benchmark. The main scheduling parameter is therefore the number of render groups. A render group owns an independent stream of frame/camera work, so increasing the number of render groups exposes parallelism across the offline image sequence rather than only within a single frame.

This distinction turned out to be important. The best-performing configuration used 64 render groups with one worker per group for the disk-output workflow, and 16 render groups with four workers per group for the in-memory workflow. Our thread-scaling results are illustrated in figure 11, showing the raster-loop throughput in figure 11a and the end-to-end throughput in figure 11b. In general, we observed better end-to-end scaling by parallelising over frame/camera render jobs than by assigning many workers to a single render group. This is expected because render-group parallelism exposes concurrency across the whole pipeline: geometry preprocessing, rasterisation, image resolve, and output handling can all proceed independently for different frames and cameras. By contrast, a single render group with many workers mostly parallelises the raster loop, leaving more of the surrounding per-frame work as serial overhead.

We fitted the measured speedup curves in figure 11 using Amdahl's law to estimate the effective parallel fraction of the raster loop and the two end-to-end workflows. The fitted parallel fractions were 97.8% for the raster loop, 99.0% for the in-memory end-to-end workflow, and 98.8% for the disk-output workflow. The fact that the end-to-end workflows have a higher fitted parallel fraction than the raster loop alone reflects the benefit of offline render-group parallelism: more of the full image-generation pipeline is being executed concurrently. This is also consistent with the comparison to the single-render-group configuration, where the fitted parallel fractions were lower at 97.7% for the in-memory workflow and 89.8% for the disk-output workflow.

The practical implication is that, for offline DIC UQ rendering, throughput is more important than the latency of any single frame. The user typically wants the complete image stack as quickly as possible, and the order in which individual frames finish is not important. The hierarchical parallelisation model in `Riley` takes advantage of this by allowing work to be distributed over independent render groups, while still retaining tile-level parallelism when it is useful. For this benchmark, the best configuration rendered the full stereo image sequence, consisting of 64 time steps and two approximately 5 MPx camera views, to disk in approximately 2.7 s. Compared with the cost of solving the FE model or subsequently running stereo DIC on the generated image sequence, this makes the synthetic image generation step a small part of the overall workflow. In other words, for this representative case, rendering is no longer the bottleneck for DIC UQ studies.

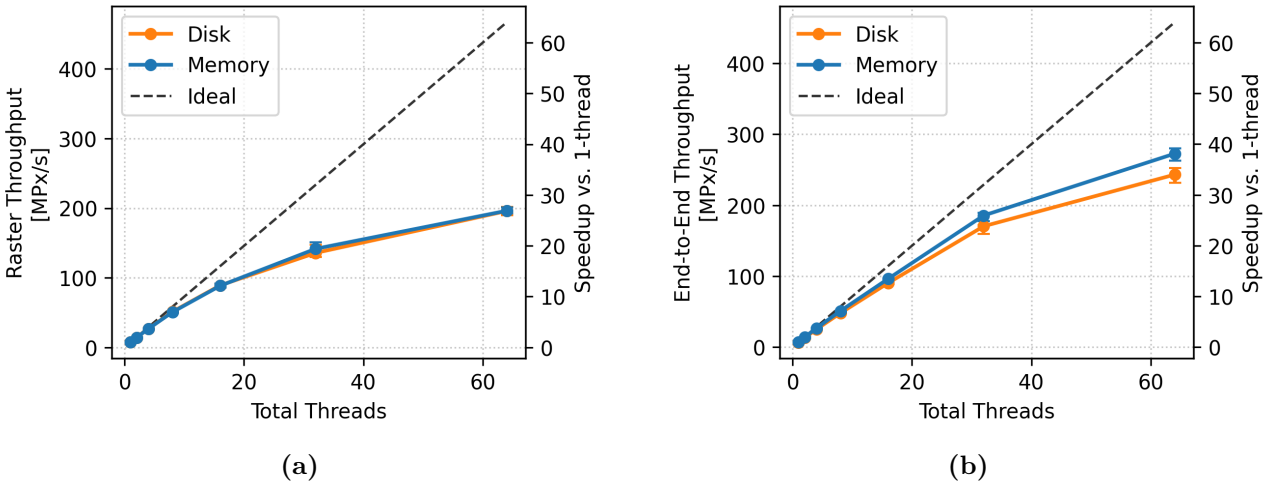


Fig. 11 Thread-scaling behaviour for the DIC UQ benchmark with offline rendering. Panel (a) shows the best raster-loop throughput by save mode, and panel (b) shows the end-to-end throughput by save mode. The right-hand axis reports speedup relative to the one thread baseline, and the dashed lines denote ideal linear scaling.

5.6 Memory scaling & practical render limits

Runtime is not the only practical constraint for DIC UQ studies. Large camera sensors, stereo image pairs, and long image sequences can also make memory usage important. In `Riley`, the largest memory contribution depends on the selected output mode. When images are returned in memory, the output array scales with the number of frames, cameras, pixels, and image channels. When images are written directly to disk, this full image stack does not need to be retained, and memory usage is dominated instead by the current frame, tile-local scratch buffers, and geometry preprocessing buffers. We approximate the total memory usage of `Riley`, M_{total}, as:

$$M_{\text{total}} \approx M_{\text{geom}} + M_{\text{frames}} + G_r W_r M_{\text{thread}} \tag{37}$$

where M_{geom} is the input mesh geometry and shader data, M_{frames} is the memory for the output frame buffer (one `f64` per pixel, per channel, per frame and per camera), G_r is the number of render groups and W_r is the number of raster workers in that group, and M_{thread} is the memory used for

thread local buffers. The thread local buffer sizes are controlled by the tile size and pixel sub-sampling with each buffer as each thread processes a tile. The user input controls the pixel sub-sampling so we use cache-aware scaling of the tile size to ensure that each thread local buffer stays within the L2 cache of 1 MB. For typical DIC UQ workflows memory usage is driven by the output frame buffer and the user specified options for how they want the final image stack output. Take for example the DIC UQ Benchmark 4 case. In this case our surface mesh consists of $31,416$ nodes and $10,472$ surface elements with 64 frames of displacement fields with 3 components (x, y, and z) and finally the uv coordinates for texture mapping. This gives a total memory usage of $M_{\text{geom}} \approx 50.18$ MB of which 48.23 MB is the displacement fields. If we are returning images in memory the output image buffer needs to hold images for two cameras with a sensor size of 2464×2056 with a single grey level image channel over 64 output frames. Our output image buffer is returned in `f64`, so the output image buffer is 5.19 GB for the in-memory mode, which completely dominates the other two terms. However, if we switch to save to disk we only need the frames we are currently working on which means $M_{\text{frames}} = G_r F_b M_{1frame}$ where F_b is the frame batch size each render group G_r is assigned and M_{1frame} is the image buffer size for single frame from one camera. If we have, for example, 4 render groups with 4 frames in a batch, our output image buffer memory drops significantly to approximately 1.3 GB. So while it is desirable to return full image stacks in memory for fast processing times users of `Riley` should be aware of the memory scaling especially for cases where sensor sizes are signifcant, there are many cameras in the scene and/or there are many frames to be rendered. Before running large synthetic image generation sweeps we recommend users undertake an approximate calculation of memory usage using eq. 37 and then a calculation of approximate simulation time using the end-to-end throughput in MPx/s given in Section 5.

5.7 Capability demonstration

Here we demonstrate the end-to-end application of `Riley` by taking the rendered images from Benchmark 4 and processing them using stereo DIC. For the DIC analysis we used the `pyvale` python package [57] with a pre-release version of the `pyvale` stereo DIC engine. The stereo DIC hardware, calibration, and analysis parameters were chosen to match the synthetic setup and are detailed in table 8, following the guidelines of the International Digital Image Correlation Society (iDICs) Good Practice Guide [13].

The end-to-end simulation-to-reconstruction workflow is shown in figure 12. First, synthetic calibration target images are generated using a standard dot grid calibration target (figures 12a and 12b) to calibrate the simulated camera stereo-pair. Next, synthetic specimen images are generated for both camera views throughout the 64-frame deformation history (figures 12c and 12d). The generated images are then processed using the `pyvale` stereo DIC engine to reconstruct the displacements. The reconstructed stereo-DIC displacements in x and y are shown in figures 12g and 12h, which visually match the ground-truth finite-element displacement fields (figures 12e and 12f) with high fidelity, demonstrating the capability of `Riley` to support end-to-end virtual calibration and DIC uncertainty quantification studies in the future. We have provided demonstration scripts for both the DIC speckle pattern rendering from the finite element mesh and the stereo calibration target rendering in the `Riley` git repository to aid users in adapting `Riley` to their own use case.

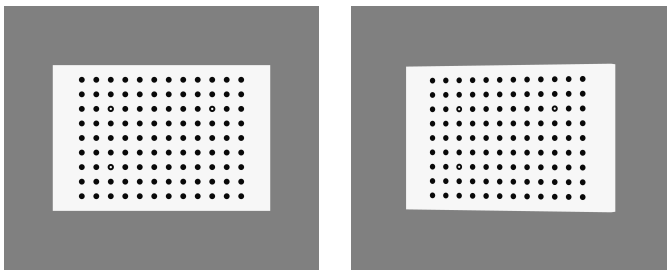
6 Conclusions

In this work we have developed `Riley`, a computational framework for rendering higher-order finite elements for digital image correlation (DIC) uncertainty quantification (UQ). We designed `Riley` to address several limitations that arise when existing rendering tools are applied to DIC UQ, including accurate rendering of higher-order finite-element geometry, support for camera distortion models, and higher-order texture reconstruction. More broadly, our aim was to keep the synthetic image generation process tied directly to the mechanics simulation a user wanted to analyse. From the development, verification, and benchmarking of `Riley`, we draw the following conclusions:

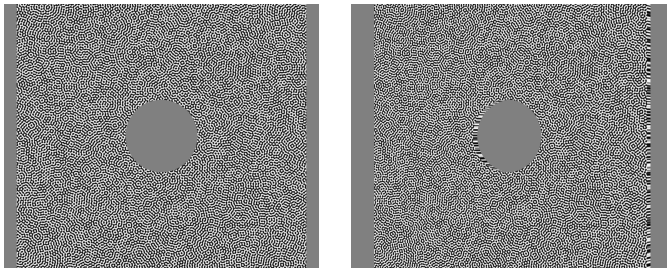
- `Riley` renders deformed finite-element surface meshes directly, including `tri3`, `tri6`, `quad4`,

Table 8: Stereo DIC parameters used for the `Riley` capability demonstration, following reporting guidlines of the International Digital Image Correlation Society (iDICs) Good Practice Guide [13].

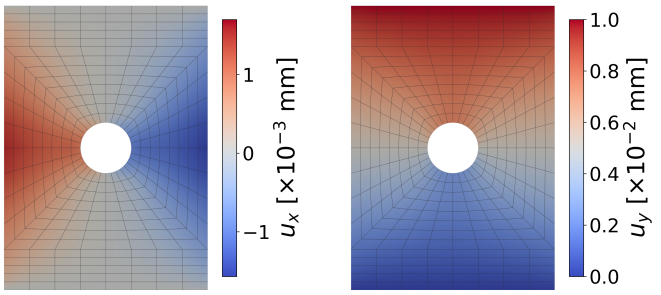
Parameter	Value / Details
Hardware & Imaging	
Camera	2× Simulated cameras (stereo pair)
Sensor resolution	2464×2056 pixels (~ 5 MPx)
Pixel size	3.45 μm
Image format	8-bit BMP
Intrinsics	
Focal length f_x, f_y	50.0 mm (14, 492.75 px)
Principal point (c_x, c_y)	(1232.0, 1028.0) px
Skew s	0.0 px
Distortion coefficients	$k_1 = k_2 = k_3 = p_1 = p_2 = 0$
Extrinsics	
Standoff distance	~ 500.0 mm
Stereo baseline	55.694 mm
Stereo angle	20.0°
Relative translation T_x, T_y, T_z	$[-54.848, 0.0, 9.671]$ mm
Relative rotation θ, ϕ, ψ	$[0.0°, 20.0°, 0.0°]$
Analysis	
DIC Software	`pyvale` (pre-release stereo engine) [57]
Match criterion	Zero-Normalized Sum of Squared Differences (ZNSSD)
Shape function	First-order (affine)
Interpolation method	Bicubic b-spline
Subset size	51×51 pixels
Step size	10 pixels



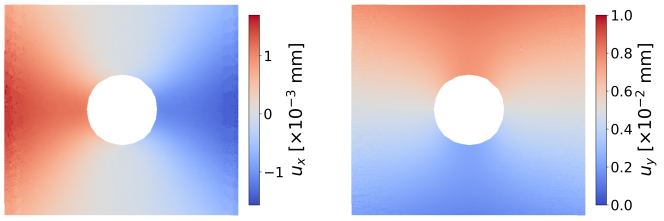
(a) Cal target C0. **(b)** Cal target C1.



(c) Specimen C0. **(d)** Specimen C1.



(e) FE u_x. **(f)** FE u_y.



(g) Stereo DIC u_x. **(h)** Stereo DIC u_y.

Fig. 12 Capability demonstration for the `Riley` stereo DIC-UQ workflow. Top row: synthetic stereo calibration target images for cameras 0 and 1. Second row: synthetic reference images of the perforated plate specimen for cameras 0 and 1. Third row: finite-element displacement fields on the specimen front face at the final frame. Bottom row: stereo-DIC displacements from the field of view at the final frame.

`quad8`, and `quad9` elements. The implementation supports mixed-element, multi-mesh, multi-camera scenes in a single rendering pipeline, which is essential for practical DIC UQ workflows.

- The projected-residual Newton pathway was verified to essentially numerical precision for admissible element geometries. Regular and affine-shear cases gave pixel-space reprojection errors below 10^{-12} px, while the more challenging midside-node bulge cases remained below approximately 10^{-10} px.
- The full rendering pipeline reproduced independently generated silhouette reference masks for all supported element types. In the single-element silhouette tests, the rendered masks matched the reference masks exactly, area errors remained no more than $3 \times 10^{-3}\%$, and the largest centroid error was still less than 4×10^{-2} px resulting purely from pixel discretisation error.
- Pixel sub-sampling behaved as expected, with rapid monotonic convergence towards the high-sub-sampling reference image and remaining differences localised to the silhouette boundary. The visibility tests also confirmed that the tile-local depth buffer correctly prevents rear-mesh contributions from leaking through closer visible geometry.
- Adaptive hulls and explicit SIMD provide large single-threaded speedups for the higher-order Newton pathways. In the spherical texture-rendering ablation, the combined adaptive-hull and SIMD path achieved raster-loop speedups of 12.99× for `tri6`, 8.92× for `quad8`, and 8.72× for `quad9`, with corresponding end-to-end speedups of 12.15×, 8.19×, and 8.02×.
- The production raster loop is already fast on a single thread, reaching 242 MPx/s for the direct `tri3` nodal pathway and roughly 56–86 MPx/s for the Newton-solved nodal pathways. For the representative threaded DIC UQ benchmark, the fitted parallel fractions were 97.8% for the raster loop, 99.0% for the in-memory end-to-end workflow, and 98.8% for the disk-output workflow.

Given the computational performance of `Riley`, combined with that of the DIC engine in the `pyvale` Python package [57], it is now possible to analyse significantly larger experimental design problems requiring DIC UQ. Take, for example, the topology optimisation problem analysed in [58]. In this type of problem, the quantity of interest is not just the mechanical performance of a candidate design, but whether that performance could be measured reliably in a real experiment. This is where fast synthetic image generation and fast DIC become especially useful: they allow the measurement process itself to be included in the design loop. More generally, `Riley` makes it practical to ask questions that would otherwise be too expensive to explore systematically. We can vary camera placement, lens parameters, speckle patterns, lighting assumptions, noise levels, and loading paths, and then quantify how these choices affect the uncertainty of the recovered DIC fields. This opens the door to simulation-driven experimental design studies where the goal is not simply to generate a plausible synthetic image, but to optimise the experiment to recover the fields of interest making sensor placement optimisation tractable for DIC. The present work provides the rendering side of this workflow. Combined with high-performance DIC analysis, it gives us a route towards large-scale virtual experiments for designing, testing, and comparing DIC measurement strategies before any physical experiment is performed. In future work, this will allow us to couple finite-element simulation, synthetic image generation, DIC, and uncertainty analysis into a single computational design loop for mechanics experiments.

Declarations

Funding

This work was supported by UKRI through the Future Leaders Fellowship scheme (grant number MR/Y015916/1).

Competing interests

The authors have no competing interests to declare that are relevant to the content of this article.

Author contributions

Lloyd Fletcher: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. Joel Hirst: Software, Validation, Writing – review & editing. Wiera Bielajewa: Software, Validation, Writing – review & editing.

Code availability

The `Riley` rasteriser source code, benchmark models, and capability scripts are made openly available under an MIT license at:
`https://github.com/Computer-Aided-Validation-Laboratory/riley-raster`.

Acknowledgements

Personal note from Lloyd Fletcher: I would like to acknowledge all contributors to the Zig programming language. `Riley` was my first substantial project in a compiled language, and Zig made this transition far more approachable than I expected. Zig was exactly the language I was looking for to implement engineering simulation tools: explicit, predictable, fast, and close enough to the hardware without hiding the details that matter. It gave me precise control over performance, memory, and computational implementation while still remaining readable and enjoyable to work with. Zig made the development of this work extremely rewarding.

References

[1] Eann A. Patterson, Sally Purdie, Richard J. Taylor, and Chris Waldon. An integrated digital framework for the design, build and operation of fusion power plants. *Royal Society Open Science*, 6(10):181847, October 2019. ISSN 2054-5703. `https://doi.org/10.1098/rsos.181847`. URL `https://doi.org/10.1098/rsos.181847`.

[2] R. Kemp, H. Lux, M. Kovari, J. Morris, R. Wenninger, H. Zohm, W. Biel, and G. Federici. Dealing with uncertainties in fusion power plant conceptual development. *Nuclear Fusion*, 57(4): 046024, March 2017. ISSN 0029-5515. `https://doi.org/10.1088/1741-4326/aa5e2c`. URL `https://doi.org/10.1088/1741-4326/aa5e2c`.

[3] Riccardo Pellegrini, Jeroen Wackers, Riccardo Broglia, Andrea Serani, Michel Visonneau, and Matteo Diez. A multi-fidelity active learning method for global design optimization problems with noisy evaluations. *Engineering with Computers*, 39(5):3183–3206, October 2023. ISSN 1435-5663. `https://doi.org/10.1007/s00366-022-01728-0`. URL `https://doi.org/10.1007/s00366-022-01728-0`.

[4] Aylar Partovizadeh, Sebastian Schöps, and Dimitrios Loukrezis. Fourier-enhanced reduced-order surrogate modeling for uncertainty quantification in electric machine design. *Engineering with Computers*, 41(4):2619–2639, August 2025. ISSN 1435-5663. `https://doi.org/10.1007/s00366-025-02123-1`. URL `https://doi.org/10.1007/s00366-025-02123-1`.

[5] Andrew Davis, Chris Waldon, Stuart I. Muldrew, Bhavin S. Patel, Patricia Verrier, Thomas R. Barrett, and Gerasimos A. Politis. Digital: accelerating the pathway. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 382(2280):20230411, August 2024. ISSN 1364-503X. `https://doi.org/10.1098/rsta.2023.0411`. URL `https://doi.org/10.1098/rsta.2023.0411`.

[6] G. Federici, L. Boccaccini, F. Cismondi, M. Gasparotto, Y. Poitevin, and I. Ricapito. An overview of the EU breeding blanket design strategy as an integral part of the DEMO design effort. *Fusion Engineering and Design*, 141:30–42, April 2019. ISSN 0920-3796. https://doi.org/10.1016/j.fusengdes.2019.01.141. URL https://www.sciencedirect.com/science/article/pii/S0920379619301590.

[7] M. Fursdon and J-H. You. Towards reliable design-by-analysis for divertor plasma facing components—Guidelines for inelastic assessment (part II: irradiated). *Fusion Engineering and Design*, 160:111831, November 2020. ISSN 0920-3796. https://doi.org/10.1016/j.fusengdes.2020.111831. URL https://www.sciencedirect.com/science/article/pii/S0920379620303793.

[8] Jeong-Ha You, Muyuan Li, and Kuo Zhang. Structural lifetime assessment for the DEMO divertor targets: Design-by-analysis approach and outstanding issues. *Fusion Engineering and Design*, 164: 112203, March 2021. ISSN 0920-3796. https://doi.org/10.1016/j.fusengdes.2020.112203. URL https://www.sciencedirect.com/science/article/pii/S0920379620307511.

[9] J. H. You, G. Mazzone, E. Visca, H. Greuner, M. Fursdon, Y. Addab, C. Bachmann, T. Barrett, U. Bonavolontà, B. Böswirth, F. M. Castrovinci, C. Carelli, D. Coccorese, R. Coppola, F. Crescenzi, G. Di Gironimo, P. A. Di Maio, G. Di Mambro, F. Domptail, D. Dongiovanni, G. Dose, D. Flammini, L. Forest, P. Frosi, F. Gallay, B. E. Ghidersa, C. Harrington, K. Hunger, V. Imbriani, M. Li, A. Lukenskas, A. Maffucci, N. Mantel, D. Marzullo, T. Minniti, A. V. Müller, S. Noce, M. T. Porfiri, A. Quartararo, M. Richou, S. Roccella, D. Terentyev, A. Tincani, E. Vallone, S. Ventre, R. Villari, F. Villone, C. Vorpahl, and K. Zhang. Divertor of the European DEMO: Engineering and technologies for power exhaust. *Fusion Engineering and Design*, 175:113010, February 2022. ISSN 0920-3796. https://doi.org/10.1016/j.fusengdes.2022.113010. URL https://www.sciencedirect.com/science/article/pii/S0920379622000102.

[10] T. R. Barrett, M. Bamford, B. Chuilon, T. Deighan, P. Efthymiou, L. Fletcher, M. Gorley, T. Grant, T. Hall, D. Horsley, M. Kovari, and M. Tindall. The CHIMERA facility development programme. *Fusion Engineering and Design*, 194:113689, September 2023. ISSN 0920-3796. https://doi.org/10.1016/j.fusengdes.2023.113689. URL https://www.sciencedirect.com/science/article/pii/S0920379623002727.

[11] J. T. Horne-Jones, M. Baxter, A. Tayeb, L. Fletcher, J. Paterson, S. Biggs-Fox, and A. Harte. Towards Virtual Qualification in Nuclear Fusion: Demonstrating Probabilistic Model Validation on a High Heat Flux Component. http://arxiv.org/abs/2605.11886, May 2026. URL http://arxiv.org/abs/2605.11886. arXiv:2605.11886 [physics.plasm-ph].

[12] Hubert Schreier, Jean-José Orteu, and Michael A. Sutton. *Image Correlation for Shape, Motion and Deformation Measurements: Basic Concepts,Theory and Applications*. Springer US, Boston, MA, 2009. ISBN 978-0-387-78746-6 978-0-387-78747-3. https://doi.org/10.1007/978-0-387-78747-3. URL https://link.springer.com/10.1007/978-0-387-78747-3.

[13] International Digital Image Correlation Society, E.M.C. Jones, and Iadicola, M.A. A Good Practices Guide for Digital Image Correlation, Edition 2. Technical report, 2025.

[14] Adel Tayeb, Lloyd Fletcher, Mike Gorley, Allan Harte, and Cory Hamelin. Image-based deformation measurements of fusion divertor armour under high heat flux loading. *The Journal of Strain Analysis for Engineering Design*, 61(4):267–283, May 2026. ISSN 0309-3247. https://doi.org/10.1177/03093247251414309. URL https://doi.org/10.1177/03093247251414309.

[15] Pascal Lava, Elizabeth M. C. Jones, Lukas Wittevrongel, and Fabrice Pierron. Validation of finite-element models using full-field experimental data: Levelling finite-element analysis data through a digital image correlation engine. *Strain*, 56(4):e12350, 2020. ISSN 1475-1305. https://doi.org/10.1111/str.12350. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/str.12350. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/str.12350.

[16] Erdogan Madenci, Amin Yaghoobi, Atila Barut, Nam Phan, Athanasios Iliopoulos, and John G. Michopoulos. Peridynamics enabled digital image correlation for tracking crack paths. *Engineering with Computers*, 39(1):517–543, February 2023. ISSN 1435-5663. https://doi.org/10.1007/s00366-021-01592-4. URL https://doi.org/10.1007/s00366-021-01592-4.

[17] Abdalrhaman Koko, Alya Abdelnour, Thorsten H. Becker, and T. James Marrow. Bridging experiments and defects' mechanics: a data-driven toolbox for configurational force analysis. *Engineering with Computers*, 42(1):21, January 2026. ISSN 1435-5663. https://doi.org/10.1007/s00366-025-02262-5. URL https://doi.org/10.1007/s00366-025-02262-5.

[18] A. Peshave, F. Pierron, P. Lava, D. Moens, and D. Vandepitte. Practical Uncertainty Quantification Guidelines for DIC-Based Numerical Model Validation. *Experimental Techniques*, 49(3): 437–457, June 2025. ISSN 1747-1567. https://doi.org/10.1007/s40799-024-00758-1. URL https://doi.org/10.1007/s40799-024-00758-1.

[19] A. Peshave, F. Pierron, P. Lava, D. Moens, and D. Vandepitte. DIC-Based Finite Element Model Validation: A Practical Case Study for In-Plane Loading of A Composite Laminate. *Experimental Mechanics*, 66(1):37–66, January 2026. ISSN 1741-2765. https://doi.org/10.1007/s11340-025-01234-6. URL https://doi.org/10.1007/s11340-025-01234-6.

[20] William L. Oberkampf and Christopher J. Roy. *Verification and Validation in Scientific Computing*. Cambridge University Press, October 2010. ISBN 978-1-139-49176-1. Google-Books-ID: 7d26zLEJ1FUC.

[21] The American Society of Mechanical Engineers (ASME). Assessing Credibility of Computational Modeling Through Verification and Validation: Application to Medical Devices. International Standard ASME V&V 40-2018, ASME, New York, USA, 2018. URL https://www.asme.org/codes-standards/find-codes-standards/v-v-40-assessing-credibility-computational-modeling-verification-validation-application-m

[22] R. Balcaen, P.L. Reu, P. Lava, and D. Debruyne. Stereo-DIC Uncertainty Quantification based on Simulated Images. *Experimental Mechanics*, 57(6):939–951, July 2017. ISSN 1741-2765. https://doi.org/10.1007/s11340-017-0288-9. URL https://doi.org/10.1007/s11340-017-0288-9.

[23] M. Rossi, P. Lava, F. Pierron, D. Debruyne, and M. Sasso. Effect of DIC Spatial Resolution, Noise and Interpolation Error on Identification Results with the VFM. *Strain*, 51(3):206–222, 2015. ISSN 1475-1305. https://doi.org/10.1111/str.12134. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/str.12134. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/str.12134.

[24] Jianhui Zhao and Bing Pan. Uncertainty quantification for 3D digital image correlation displacement measurements using Monte Carlo method. *Optics and Lasers in Engineering*, 170:107777, November 2023. ISSN 0143-8166. https://doi.org/10.1016/j.optlaseng.2023.107777. URL https://www.sciencedirect.com/science/article/pii/S0143816623003068.

[25] Satoru Yoneyama and Keisuke Iizuka. Generation of Simulation Images Based on Finite Element Results for Performance Evaluation of Digital Image Correlation. *Advanced Experimental Mechanics*, 8:53–61, 2023. https://doi.org/10.11395/aem.8.0_53.

[26] Michel Bornert, Pascal Doumalin, Jean-Christophe Dupré, Christophe Poilane, Laurent Robert, Evelyne Toussaint, and Bertrand Wattrisse. Shortcut in DIC error assessment induced by image interpolation used for subpixel shifting. *Optics and Lasers in Engineering*, 91:124–133, April 2017. ISSN 0143-8166. https://doi.org/10.1016/j.optlaseng.2016.11.014. URL https://www.sciencedirect.com/science/article/pii/S0143816616304511.

[27] Yong Su, Qingchuan Zhang, Xiaohai Xu, Zeren Gao, and Shangquan Wu. Interpolation bias for the inverse compositional Gauss–Newton algorithm in digital image correlation. *Optics and Lasers in Engineering*, 100:267–278, January 2018. ISSN 0143-8166. https://doi.org/10.1016/j.optlaseng.2017.09.013. URL https://www.sciencedirect.com/science/article/pii/S014381661730341X.

[28] R. Balcaen, P. L. Reu, P. Lava, and D. Debruyne. Influence of Camera Rotation on Stereo-DIC and Compensation Methods. *Experimental Mechanics*, 58(7):1101–1114, September 2018. ISSN 1741-2765. https://doi.org/10.1007/s11340-017-0368-x. URL https://doi.org/10.1007/s11340-017-0368-x.

[29] R. Balcaen, P. L. Reu, and D. Debruyne. Stereo-DIC Uncertainty Estimation Using the Epipolar Constraint and Optimized Three Camera Triangulation. *Experimental Techniques*, 42(1):115–120, February 2018. ISSN 1747-1567. https://doi.org/10.1007/s40799-017-0207-0. URL https://doi.org/10.1007/s40799-017-0207-0.

[30] H. Cui, Z. Zeng, H. Zhang, and F. Yang. Effect of Speckle Edge Characteristics on DIC Calculation Error. *Experimental Mechanics*, 64(7):1143–1160, September 2024. ISSN 1741-2765. https://doi.org/10.1007/s11340-024-01078-6. URL https://doi.org/10.1007/s11340-024-01078-6.

[31] Frédéric Sur, Benoît Blaysat, and Michel Grédiac. On biases in displacement estimation for image registration, with a focus on photomechanics - Extended version. Research Report, LORIA (Université de Lorraine, CNRS, INRIA) ; Institut Pascal (Université Clermont-Auvergne, SIGMA, CNRS), June 2020. URL https://hal.science/hal-02862808.

[32] R. Balcaen, L. Wittevrongel, P. L. Reu, P. Lava, and D. Debruyne. Stereo-DIC Calibration and Speckle Image Generator Based on FE Formulations. *Experimental Mechanics*, 57(5):703–718, June 2017. ISSN 1741-2765. https://doi.org/10.1007/s11340-017-0259-1. URL https://doi.org/10.1007/s11340-017-0259-1.

[33] D. P. Rohe and E. M. C. Jones. Generation of Synthetic Digital Image Correlation Images Using the Open-Source Blender Software. *Experimental Techniques*, 46(4):615–631, August 2022. ISSN 1747-1567. https://doi.org/10.1007/s40799-021-00491-z. URL https://doi.org/10.1007/s40799-021-00491-z.

[34] Fabrice Pierron. Material Testing 2.0: A brief review. *Strain*, 59(3):e12434, 2023. ISSN 1475-1305. https://doi.org/10.1111/str.12434. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/str.12434. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/str.12434.

[35] F. Pierron and M. Grédiac. Towards Material Testing 2.0. A review of test design for identification of constitutive parameters from full-field measurements. *Strain*, 57(1):e12370, 2021. ISSN 1475-1305. https://doi.org/10.1111/str.12370. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/str.12370. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/str.12370.

[36] Michelangelo Marsala, Angelos Mantzaflaris, Bernard Mourrain, Sam Whyman, and Mark Gammon. From CAD to representations suitable for isogeometric analysis: a complete pipeline. *Engineering with Computers*, 40(6):3429–3447, December 2024. ISSN 1435-5663. https://doi.org/10.1007/s00366-024-02065-0. URL https://doi.org/10.1007/s00366-024-02065-0.

[37] Han Zhao, John T. Hwang, and Jiun-Shyan Chen. Open-source shape optimization for isogeometric shells using FEniCS and OpenMDAO. *Engineering with Computers*, 41(3):1877–1899, June 2025. ISSN 1435-5663. https://doi.org/10.1007/s00366-025-02116-0. URL https://doi.org/10.1007/s00366-025-02116-0.

[38] Ivo Babuška and Manil Suri. The p and h-p Versions of the Finite Element Method, Basic Principles and Properties. *SIAM Review*, 36(4):578–632, December 1994. ISSN 0036-1445. https://doi.org/10.1137/1036141. URL https://epubs.siam.org/doi/10.1137/1036141.

[39] A. Düster, A. Niggl, V. Nübel, and E. Rank. A Numerical Investigation of High-Order Finite Elements for Problems of Elastoplasticity. *Journal of Scientific Computing*, 17(1):397–404, December 2002. ISSN 1573-7691. https://doi.org/10.1023/A:1015189706770. URL https://doi.org/10.1023/A:1015189706770.

[40] O. C. Zienkiewicz, R. L. Taylor, and David Fox. *The Finite Element Method for Solid and Structural Mechanics*. Butterworth-Heinemann, November 2013. ISBN 978-1-85617-634-7 978-0-08-095136-2. https://doi.org/10.1016/C2009-0-26332-X. URL https://www.sciencedirect.com/book/monograph/9781856176347/the-finite-element-method-for-solid-and-structural-mechanics.

[41] Marcus D. Hanwell, Kenneth M. Martin, Aashish Chaudhary, and Lisa S. Avila. The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards. *SoftwareX*, 1-2:9–12, September 2015. ISSN 2352-7110. https://doi.org/10.1016/j.softx.2015.04.001. URL https://www.sciencedirect.com/science/article/pii/S2352711015000035.

[42] William J. Schroeder, Francois Bertel, Mathieu Malaterre, David Thompson, Philippe P. Pebay, Robert O'Bara, and Saurabh Tendulkar. Methods and Framework for Visualizing Higher-Order Finite Elements. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):446–460, July 2006. ISSN 1077-2626. https://doi.org/10.1109/TVCG.2006.74. URL https://doi.org/10.1109/TVCG.2006.74.

[43] D. F. Wiley, H. R. Childs, B. Hamann, and K. I. Joy. *Ray Casting Curved-Quadratic Elements*. The Eurographics Association, 2004. ISBN 978-3-905673-07-4. URL https://doi.org/10.2312/VisSym/VisSym04/201-210.

[44] Jean-François Remacle, Nicolas Chevaugeon, Émilie Marchandise, and Christophe Geuzaine. Efficient visualization of high-order finite elements. *International Journal for Numerical Methods in Engineering*, 69(4):750–771, 2007. ISSN 1097-0207. https://doi.org/10.1002/nme.1787. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.1787. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.1787.

[45] Rémi Feuillet, Matthieu Maunoury, and Adrien Loseille. On pixel-exact rendering for high-order mesh and solution. *Journal of Computational Physics*, 424:109860, January 2021. ISSN 0021-9991. https://doi.org/10.1016/j.jcp.2020.109860. URL https://www.sciencedirect.com/science/article/pii/S0021999120306343.

[46] Blake Nelson, Eric Liu, Robert M. Kirby, and Robert Haimes. ElVis: A System for the Accurate and Interactive Visualization of High-Order Finite Element Solutions. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2325–2334, December 2012. ISSN 1077-2626. https://doi.org/10.1109/TVCG.2012.218. URL http://ieeexplore.ieee.org/document/6327237/.

[47] J. Peiro, D. Moxey, B. Jordi, S. J. Sherwin, B. W. Nelson, R. M. Kirby, and R. Haimes. High-Order Visualization with ElVis. In Norbert Kroll, Charles Hirsch, Francesco Bassi, Craig Johnston, and Koen Hillewaert, editors, *IDIHOM: Industrialization of High-Order Methods - A Top-Down Approach: Results of a Collaborative Research Project Funded by the European Union, 2010 - 2014*, pages 521–534. Springer International Publishing, Cham, 2015. ISBN 978-3-319-12886-3. https://doi.org/10.1007/978-3-319-12886-3_24. URL https://doi.org/10.1007/978-3-319-12886-3_24.

[48] Frédéric Sur, Benoît Blaysat, and Michel Grédiac. Rendering Deformed Speckle Images with a Boolean Model. *Journal of Mathematical Imaging and Vision*, 60(5):634–650, June 2018. ISSN 0924-9907, 1573-7683. https://doi.org/10.1007/s10851-017-0779-4. URL http://link.springer.com/10.1007/s10851-017-0779-4.

[49] P. L. Reu, E. Toussaint, E. Jones, H. A. Bruck, M. Iadicola, R. Balcaen, D. Z. Turner, T. Siebert, P. Lava, and M. Simonsen. DIC Challenge: Developing Images and Guidelines for Evaluating Accuracy and Resolution of 2D Analyses. *Experimental Mechanics*, 58(7):1067–1099, September 2018. ISSN 1741-2765. https://doi.org/10.1007/s11340-017-0349-0. URL https://doi.org/10.1007/s11340-017-0349-0.

[50] P. L. Reu, B. Blaysat, E. Andó, K. Bhattacharya, C. Couture, V. Couty, D. Deb, S. S. Fayad, M. A. Iadicola, S. Jaminion, M. Klein, A. K. Landauer, P. Lava, M. Liu, L. K. Luan, S. N. Olufsen, J. Réthoré, E. Roubin, D. T. Seidl, T. Siebert, O. Stamati, E. Toussaint, D. Turner, C. S. R. Vemulapati, T. Weikert, J. F. Witz, O. Witzel, and J. Yang. DIC Challenge 2.0: Developing Images and Guidelines for Evaluating Accuracy and Resolution of 2D Analyses. *Experimental Mechanics*, 62(4):639–654, April 2022. ISSN 1741-2765. https://doi.org/10.1007/s11340-021-00806-6. URL https://doi.org/10.1007/s11340-021-00806-6.

[51] L. Fletcher, J. Hirst, and W. Bielajewa. Riley: Zig software rasteriser for finite-element image synthesis. https://github.com/Computer-Aided-Validation-Laboratory/riley-raster, 2026. URL https://github.com/Computer-Aided-Validation-Laboratory/riley-raster.

[52] Indrajeet Sahu. Bilinear-inverse-mapper: Analytical solution and algorithm for inverse mapping of bilinear interpolation of quadrilaterals. *Advances in Engineering Software*, 208:103975, October 2025. ISSN 0965-9978. https://doi.org/10.1016/j.advengsoft.2025.103975. URL https://www.sciencedirect.com/science/article/pii/S0965997825001139.

[53] Andrew Kelly. Zig Programming Language. https://ziglang.org/download/, April 2026. URL https://ziglang.org/download/.

[54] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009. ISSN 1097-0207. https://doi.org/10.1002/nme.2579. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2579. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2579.

[55] Cody J. Permann, Derek R. Gaston, David Andrš, Robert W. Carlsen, Fande Kong, Alexander D. Lindsay, Jason M. Miller, John W. Peterson, Andrew E. Slaughter, Roy H. Stogner, and Richard C. Martineau. MOOSE: Enabling massively parallel multiphysics simulation. *SoftwareX*, 11:100430, January 2020. ISSN 2352-7110. https://doi.org/10.1016/j.softx.2020.100430. URL https://www.sciencedirect.com/science/article/pii/S2352711019302973.

[56] Tianchen Hu and Mark C. Messner. A Simple, Scalable Large Deformation Solid Mechanics Implementation in the MOOSE Framework. *ACM Transactions on Mathematical Software*, 51 (1):4:1–4:22, April 2025. ISSN 0098-3500. https://doi.org/10.1145/3716308. URL https://dl.acm.org/doi/10.1145/3716308.

[57] Joel Hirst, Lorna Sibson, Adel Tayeb, Ben Poole, Megan Sampson, Wiera Bielajewa, Michael Atkinson, Alex Marsh, Rory Spencer, Rob Hamill, Cory Hamelin, Allan Harte, and Lloyd Fletcher. PYVALE: A Fast, Scalable, Open-Source 2D Digital Image Correlation (DIC) Engine Capable of Handling Gigapixel Images. http://arxiv.org/abs/2601.12941, January 2026. URL http://arxiv.org/abs/2601.12941. arXiv:2601.12941 [eess.IV].

[58] Rory Spencer, Lloyd Fletcher, Rob Hamill, Cory Hamelin, and Allan Harte. Design of a Materials Testing 2.0 Creep Test using the Virtual Fields Method and Open Source Tools. *Experimental Mechanics*, 2026. Under Review.