



## GPU-Accelerated Post-Processing and Animated Volume Rendering of Isogeometric Analysis Results

Harshil Shah<sup>1</sup>, Xin Huang<sup>1</sup>, Onur Rauf Bingol<sup>1</sup>, Manoj R. Rajanna<sup>1</sup>, Adarsh Krishnamurthy<sup>1</sup> 

<sup>1</sup>Iowa State University,

Corresponding author: Adarsh Krishnamurthy, [adarsh@iastate.edu](mailto:adarsh@iastate.edu)

**Abstract.** Isogeometric analysis (IGA) has enabled better CAD integration by using the same spline representations for modeling and analysis. Traditionally, the finite element analysis results are visualized by creating a texture map of the property of interest and superimposing them over the boundary representation (B-rep) model or the mesh. This technique cannot be directly used to render internal quantities of interest without computationally intensive sectioning and remapping of the textures. In this paper, we present a GPU accelerated algorithm that produces a time-varying voxelized representation of the property of interest. We then render the voxelized models generated using this approach at an interactive frame rate. To voxelize the models, we perform a modified ray intersection test with Bézier elements using a localized ray grid. We then generate a variable density voxel model representing the analysis results using the intersection data, which is repeated for the different time frames of the analysis. The complete time-series data is stored in GPU memory using a flat data structure, which is then rendered using GPU-accelerated ray casting. This direct voxelization technique enables a detailed analysis of the simulation using interactive slicing techniques without computationally intensive post-processing. We demonstrate this approach for two biomechanics simulations—a cardiac solid mechanics model and an aorta fluid dynamics model. These models were tested at different resolutions using single or batch frame processing. Our method can render the results of a complete isogeometric analysis at an interactive frame rate of over 30 frames per second for all test cases.

**Keywords:** Ray intersection, isogeometric analysis post-processing, volumetric splines, GPU-accelerated geometric algorithms, animated volume rendering.

**DOI:** <https://doi.org/10.14733/cadaps.2022.779-796>

## 1 INTRODUCTION

The development of isogeometric analysis (IGA) [4, 8] has enabled tighter integration of engineering design and computational analysis. The core idea of IGA is to use the same basis functions for the representation of geometry in CAD and the approximation of solution fields in finite element analysis (FEA). IGA eliminates the tedious mesh generation process from complex CAD models while also improving the solution quality by incorporating smooth basis functions for engineering analysis [5]. However, the solution fields of IGA are calculated on the control points of the underlying spline geometry, which may not lie on the actual geometry of the CAD model. This makes the rendering of the solution fields challenging since the solution fields also need to be evaluated on the geometry locations using the same analysis and geometry basis functions.

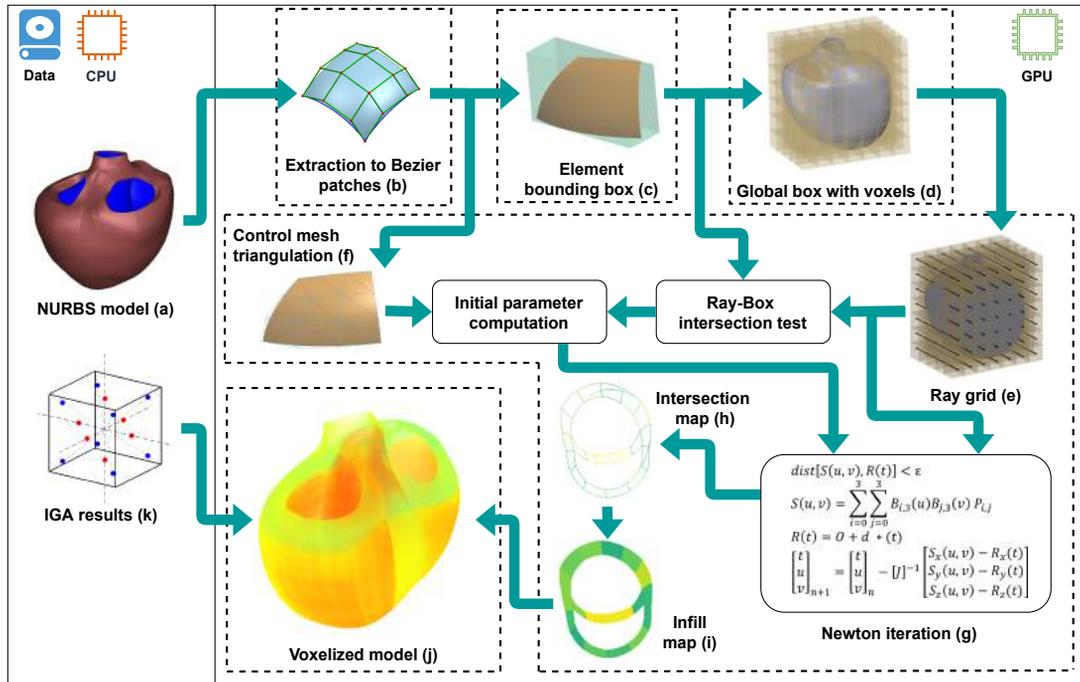
Visualizing the simulation results of isogeometric analyses can serve as valuable feedback to help the designer understand the effect of design changes. This is especially relevant if a solution field, such as in-plane strain, is used as a metric for evaluating the design. After solving the IGA simulation, the control variables (or degrees of freedom) for the solution fields (e.g., displacement, velocity, temperature) are computed on the control points, typically not located on the physical geometry. These need to be weighted by the basis functions to generate continuous solution fields mapped to the physical geometry. Sophisticated visualization techniques, such as direct volume rendering [17, 20], isosurface mesh extraction [18, 2], and direct rendering of isosurfaces [12, 22, 19] have been developed for visualizing volumetric IGA results. However, these methods are usually computationally intensive, making them difficult to interactively visualize the results of a dynamic structural analysis simulation or fluid-structure interaction simulations on volumetric splines.

Conventional mesh rendering uses boundary representation (B-Rep) to represent solid models using a set of faces that bound the model. Current methods for rendering and interacting with 3D volume data rely on rendering the CAD model's surfaces and use texture mapping to visualize quantities of interest. However, this rendering method does not facilitate the display of quantities of interest, such as stresses and strains that lie inside the volume of the geometry. A key challenge in interpreting dynamic simulation results is visualizing the time-varying quantities of interest interactively. This requires a visualization system that exhibits two main features: render sizeable time-varying data sets or 3D volume data interactively and allow the user to interact with these data sets to provide real-time feedback from the simulation results.

Ray-casting is usually used to render volume data and is computationally more intensive than rasterization. Performing ray casting with volumetric splines used in IGA is still computationally intensive to perform interactively. In this work, we first voxelize the isogeometric mesh using a GPU-accelerated ray intersection algorithm for cubic-Bézier volumes to convert volumetric splines to time-varying voxelized data structures. These data structures intuitively mirror the data structures required for 3D textures used on the GPU for sampling the 3D volume. We then use GPU ray casting to volume render the frames of the time-varying simulation. This makes the volume rendering of dynamic IGA simulation results interactive, allowing interactive manipulation in the 3D environment.

One specific post-processing application of IGA results that we focus on in this paper is IGA using cubic-Hermite elements. Cubic-Hermite finite element interpolation schemes have been popular because of their convergence properties in finite element simulations [3, 14] and their ability to capture smooth geometries compactly. Cubic-Hermite volumes have a one-to-one mapping to cubic-Bézier volumes. Similarly, Bézier extraction can be used to extract Bézier volumes from IGA that use non-uniform rational B-splines (NURBS). We demonstrate our method's applicability by using this approach to render the flow visualization in a fluid-structure interaction (FSI) simulation of blood flow in the aortic root.

In this paper, we present an approach to render IGA simulations that a user can interact with while visualizing the animated results. Our method works with dynamic simulations with moving untrimmed NURBS volumes. In our approach, we first voxelize volumetric splines by using a GPU-accelerated ray intersection method (see Figure 1). We use a modified Newton-Raphson root-finding method (referred to as the *Newton method* in this paper) that incorporates the parametric ray equations and the spline equations to find the



**Figure 1:** Different steps for obtaining animated volume rendering of isogeometric analysis. The steps are divided between CPU and GPU processes; each dashed line bounding box in the GPU section represents individual kernels executed on the GPU.

intersection point to within user-defined accuracy. Identifying initial parameters close to the solution is critical for the Newton method to find the intersections accurately. We compute these initial parameters by performing intersections of the ray with the control mesh. The voxel that contains the intersection point is then identified and assigned an index corresponding to the parent element of the surface patch. Once all the boundary voxels have been identified, we perform an infill operation along the ray. Finally, the IGA result is assigned to the voxel, based on the result value at the Gauss point closest to the voxel center. The voxel models for all the timesteps are computed and stored in the GPU memory to be rendered and visualized interactively. After voxelizing all the timesteps, we use a GPU-accelerated volume rendering method to animate the results of the simulations interactively. Using our method, we can render the results of complex simulations interactively at over 30 frames-per-second (fps) while simultaneously interacting with the animation by performing dynamic rotation and slicing operations.

Figure 1 shows the different steps of our approach, along with the breakdown of the different processes between the CPU and the GPU. Section 3 outlines approach for ray intersection on NURBS volumes. It consist of Bézier decomposition (b), bounding box computation (c-d), compute ray grid (e) and control mesh triangulation (f). Section 4 outlines the direct voxelization method with infill and assigning of the IGA result field value to the voxelized model. The section includes Newton method test (g), compute of boundary voxels (h) and infilled voxel model (i). We combine the IGA results (k) with infill model and generated the visualization model (j). Section 5 describes the process for the animated rendering of variable density voxel models obtained after assigning the IGA result field. Finally, Section 6 details the two different types of IGAs used to demonstrate the proposed approach and the timing data for different resolutions.

## 2 RELATED WORK

There have been many research studies on visualizing the results of finite-element analysis and IGA. We present a few critical related works that are directly applicable to our approach. An approximate mesh-based methodology for visualizing IGA results of shell structures was proposed by Hsu et al. [7]. They constructed a visualization mesh. The mesh points' coordinates were used to find their closest points on the NURBS surface and their parametric coordinates. The solution values were evaluated and then transferred back to the visualization mesh points along with control variables, control points, and basis functions information. However, such visualization techniques cannot be directly used with volumetric splines. Calculating the internal quantities of interests, such as stresses and strains, requires some form of ray-marching or intersection approach.

One of the early algorithms to compute ray intersection with surfaces used the method of computing roots of a polynomial [10]. The Newton iteration technique to compute all the roots for the ray-surface combination was explored by Toth [30]. Kumar and Manocha [15] developed a method to decompose the NURBS surfaces into Bézier patches and then perform the triangulation of the surface for rendering. The Newton approach was improved by Geimer and Abert [6] by not requiring the surface triangulation. They also performed a decomposition operation where a higher degree NURBS surface is decomposed into cubic Bézier surface patches. A related approach is to compute approximate surface intersection by using surface bounding-boxes [1, 21]. However, the intersection point obtained using the bounding-box approaches may not lie precisely on the surface.

The GPUs available during the previous researches were not powerful enough to have a significant advantage over CPU computations. Intersection and rendering problems gained momentum after the development of dedicated GPUs. As programming graphic hardware became accessible, faster algorithms were developed for intersection and rendering. Purcell et al. [26] developed an algorithm that implements a multilevel intersection test on the triangulated surface. Their implementation showed that architecture for real-time ray tracing is not fundamentally different from the conventional CPU algorithm.

The computation of the initial guess is crucial for any Newton iteration method since the number of iterations required for convergence depends on it. Several approaches, such as ray intersection with trapezoid prism generated by the control mesh [13] and subdivision of the convex hull of the control mesh into multiple bounding boxes [24] have been proposed to compute the initial parameter. Another method proposed by Shen et al. [29] uses a matrix-based implicit representation of Bézier patches (called M-rep [16]) to compute the intersection between a line and the NURBS surface. Based on their data, the computation time of their method is comparatively higher than that of the Newton iteration but has better accuracy.

Visualizing volumetric data has been a challenging problem in computer graphics, and several approaches have been proposed; we highlight some works closely related to ours. Ray tracing approaches to render variable volume density analytic functions that include specific absorptivity and emissivity have been explored before [11]. Volumetric analysis and visualization can also help in virtual simulations. Rushmeier and Hamins [28] used volume visualization to simulate Heptane pool fire to understand the radiation due to various components. Voxel methods inherently deal with a large amount of data and might require specialized architectures for rendering [27]. Another field where volume rendering is extensively used is medical imaging, where volume data obtained by either CT or MR imaging needs to be visualized [34]. These approaches also focus on interpolating the data to render smooth fields from low-resolution input. Young and Krishnamurthy [33] developed an algorithm to create multilevel voxelization of B-rep models and interactively render them. Their approach uses a different resolution for the boundary voxels and interior voxels, which allows for high resolution on the outside surface while the inner volume being homogeneous, occupies less memory. However, most previous approaches do not deal with time-varying or animated volume rendering, which we directly address in our framework.

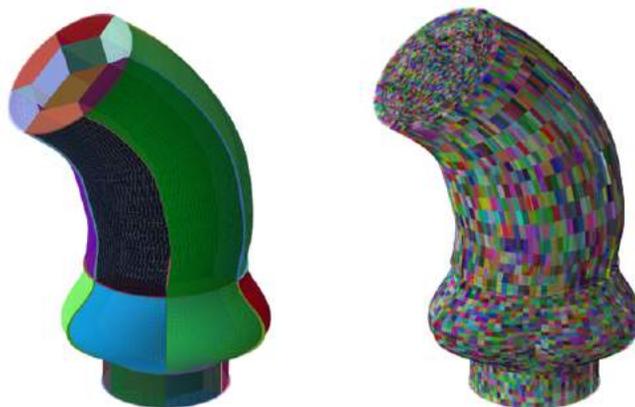
### 3 RAY INTERSECTION WITH NURBS ELEMENTS

This section outlines the process for obtaining the intersection data for a model consisting of NURBS elements using the modified ray-intersection test using Newton method. The NURBS data consisting of either a sequence of control points or Hermite elements is processed and stored in the CPU memory. This volume element data is further decomposed to produce Bézier surface elements for each of the six faces of the NURBS element. We then use a bounding volume hierarchy to cull non-intersecting rays with the individual surface elements. After the culling operation, we perform the Newton iteration method to obtain the surface patch intersection points within a user-defined tolerance. These surface intersection points are then processed to perform the element infill operation which is explained in Section 4.

#### 3.1 Preprocessing

Running a ray intersection test directly on the NURBS volume element is computationally intensive due to non-uniform knot vectors and rational surfaces. To simplify this process, the six faces of each NURBS element are decomposed into Bézier patches (Figure 1(b)). We perform Bézier decomposition using A5.7 algorithm from the NURBS Book [25]. The algorithm uses the knot refinement method to break the NURBS knot vector into Bézier knot vectors and recompute control point mesh for individual Bézier patches. The knot vector for the decomposed Bézier surface element is of the form shown in Equation 1, where  $p$  is the element's degree. Figure 2 shows the NURBS Aorta model (on the left), which is decomposed into Bézier patches (on the right). We then compute axis-aligned bounding boxes for each Bézier surface patch (Figure 1(c)). The individual element boxes are combined to form a global bounding box for the model (Figure 1(d)).

$$\text{knot vector} = [0_0, \dots, 0_p, 1_{p+1}, \dots, 1_{2p+1}] \quad (1)$$

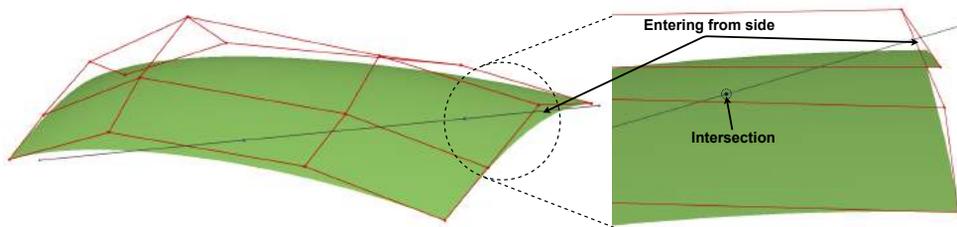


**Figure 2:** Bézier decomposition of the Aorta model showing the NURBS elements (on the left) and the Bézier surface patches (on the right).

We create a voxel grid around the model (Figure 1(d)) based on the global bounding box using a user-defined voxelization resolution. To compute the ray grid, we select the centers of each voxel on the lowest plane as the ray origin. This ray grid is different from the pixel resolution used for rendering; it is only used for the voxelization. We expand the bounding box by one voxel size in both directions perpendicular to the ray direction to prevent missing any part of the model. Figure 1(e) shows an example of a ray grid around a model with voxel centers as the ray origin. We select the direction of the ray as one of the principal directions of the coordinate system.

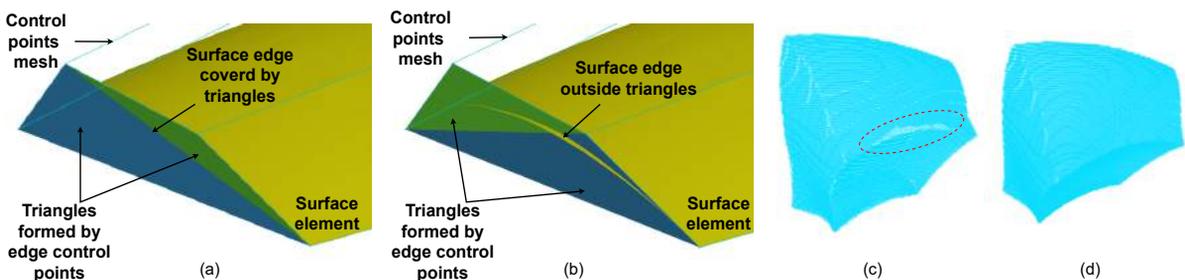
### 3.2 Ray Intersection Culling

To reduce the ray-surface computations, we perform two culling operations. First, we cull the rays not intersecting with the surface patch bounding box. The bounding-box culling is performed using a ray box intersection test (please see [23] for details). The bounding box is dilated to accommodate any planar elements with zero thickness. Second, we cull the rays not intersecting with the control mesh triangulation. Figure 14 shows the triangulation of the control mesh, which is used both for the culling operation (Equation 2) as well as to compute the initial parameters for the Newton iteration (see Section 3.3 for details). However, certain rays might not intersect with the Bézier patch control mesh triangulation but intersect the surface. The ray might enter the gap between the control mesh edge and the surface edge as shown in Figure 3. To deal with such rays, we use additional intersection tests with the edge triangles of the control mesh.



**Figure 3:** Using only the control mesh triangle intersections to perform the culling operation might miss intersections with the surface. In this example, the ray enters from the side gap between control points mesh and surface patch.

Similar to control mesh triangulation, we can form edge triangles using the edge control points. Adding these additional triangles will ensure that any ray intersecting with the control mesh and the edge triangles will intersect the surface patch. However, there exist several ways to construct the edge triangles for surface patches with more than three control points along an edge. We need to select a combination of triangles that will ensure that the edge is enclosed completely, as shown in Figure 4 (a). Automating this process requires computing the convex hull of the edge control points, which is computationally intensive. Alternatively, we perform the edge triangle intersection test on all possible edge triangles. While this might result in repetitive

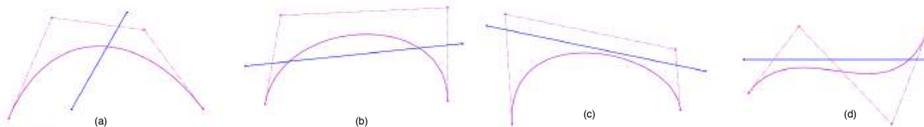


**Figure 4:** Using edge triangles from edge control points to prevent missing intersections. In this example, for a  $4 \times 4$  bi-cubic element, a particular selection of the triangle pair can result in some part of the element not being completely enveloped (b), which might result in missing intersection points for that part of the element. Missing intersections (c) near the surface edge due to the gap between the surface and the control mesh and using side triangles intersection test (d) to prevent missing intersections.

intersections being found at the same location, it ensures that we do not miss any surface intersection. Figure 4 (c) and (d) shows the intersection points with the surface patches of a single volume element with and without the edge triangles intersection test respectively. The edge triangle intersection along with the control mesh triangle intersection ensures that any ray intersecting with the Bézier surface element will intersect with at least one triangle.

### 3.3 Computing Intersections Using Newton Method

We perform the ray surface intersection test on any ray that is not culled using the bounding box intersection test, the control mesh triangle intersection test, or the edge triangle intersection test. We use a modified Newton method, an iterative root-finding algorithm, to compute the ray intersection test with a Bézier surface patch. For any iterative root-finding algorithm, starting from a close initial guess reduces the number of iterations required for convergence. To obtain a good estimate of the initial parameter, we compute intersection with the triangulation of control points mesh and the edge triangles. Since the control point mesh may not be planar, we triangulate it to perform the intersection operation. We assume that the number of intersections with the control points mesh as a reasonable estimate for the number of intersections with the element and subsequent initial parameter computation (Figure 5).



**Figure 5:** Some of the possible cases of number of intersection with parametric curve and control points grid shown as [curve, control points]: (a) [1,1], (b) [2,2], (c) [0, 2], (d) [1,3].

To check for intersections with triangles, we use the modified line plane intersection test [9]. The control mesh triangulation is the same for each Bézier patch of a given degree. Equation 2 represents formulation for the intersection test with a plane defined by two vectors  $P_{01}$  and  $P_{02}$ . These two vectors are formed by the triangle points  $P_0, P_1, P_2$ , where  $P_0$  is taken as the origin to compute  $u$  and  $v$  parameters in the Barycentric coordinate system.  $l_{ab}$  is the direction vector for the ray which is selected as one of the principal direction of the coordinate system. We select the center of the voxel as the ray start point  $l_a$ . The two conditions shown in Equation 2 need to be satisfied to obtain a valid ray-triangle intersection. The intersection point parameters are then scaled to the Bézier element coordinate system.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \begin{bmatrix} (-l_{ab})_x & (P_{01})_x & (P_{02})_x \\ (-l_{ab})_y & (P_{01})_y & (P_{02})_y \\ (-l_{ab})_z & (P_{01})_z & (P_{02})_z \end{bmatrix}^{-1} \begin{bmatrix} (l_a - P_0)_x \\ (l_a - P_0)_y \\ (l_a - P_0)_z \end{bmatrix} \quad (2)$$

$$\begin{aligned} u, v &\geq 0 \\ u + v &\leq 1 \end{aligned}$$

Using the initial parameters obtained from the control mesh triangle or edge triangle intersection, we apply the iterative root-finding method to obtain intersections with the Bézier element. Since we can differentiate the Bézier surface equations, we can directly compute the Jacobian used in the modified Newton method. Since the end goal of our approach is to voxelize the solution field, it is convenient to use a parametric ray  $R(t)$ , with  $t$  as a 1D parameter in our formulation. The objective function is the distance between the computed

surface point  $S(u, v)$  and the ray point  $R(t)$ . Equation 3 shows the expression to obtain both the points based on the surface and ray parameters ([25]). Here  $B$ s are Basis functions, and  $P$ s are control points for the Bézier element.  $o$  is the origin of the ray which is the center of the voxel.  $d$  is the unit direction vector for the ray.

$$\begin{aligned} \text{dist} [S(u, v), R(t)] &< \varepsilon \\ S(u, v) &= \sum_{i=0}^{P_u} \sum_{j=0}^{P_v} B_{i,P_u}(u) B_{j,P_v}(v) P_{i,j} \\ R(t) &= o + d * (t) \end{aligned} \quad (3)$$

Equation 4 is the modified Newton method for obtaining the roots for Equation 3. As shown in Equation 3, the criteria for convergence is the value  $\varepsilon$ , which can be user-defined. There is a possibility of a ray intersecting the control mesh and not the surface, as shown in Figure 5(c). We use two termination conditions to ensure that the Newton iteration does not run indefinitely. If the updated  $u, v$  parameters are outside the bounds of  $[0, 1]$ , the parameter values are reassigned to a random value between  $[0, 1]$ . If the reassignment process is performed more than a predefined number of times, then the Newton method is terminated. The second termination criteria is an upper bound on the number of iterations without convergence. We empirically observed that the converged intersection point is obtained within 10 iterations for our test models. Hence we set this value to be 20 in our implementation. Once the intersection point is computed, we use the ray  $t$  parameter for the voxelization and infill operation explained in the next section.

$$\begin{aligned} \begin{bmatrix} t \\ u \\ v \end{bmatrix}_{n+1} &= \begin{bmatrix} t \\ u \\ v \end{bmatrix}_n - [J]^{-1} \begin{bmatrix} S_x(u, v) - R_x(t) \\ S_y(u, v) - R_y(t) \\ S_z(u, v) - R_z(t) \end{bmatrix} \\ \text{where } J &= \begin{bmatrix} -\frac{dR_x(t)}{dt} & \frac{dS_x(u, v)}{du} & \frac{dS_x(u, v)}{dv} \\ -\frac{dR_y(t)}{dt} & \frac{dS_y(u, v)}{du} & \frac{dS_y(u, v)}{dv} \\ -\frac{dR_z(t)}{dt} & \frac{dS_z(u, v)}{du} & \frac{dS_z(u, v)}{dv} \end{bmatrix} \end{aligned} \quad (4)$$

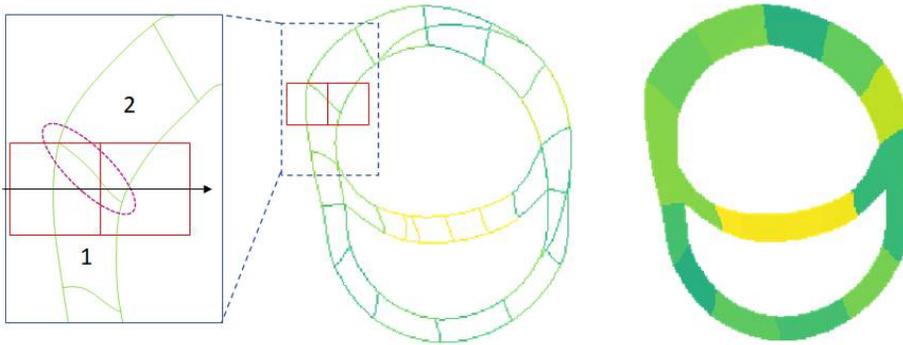
## 4 VOXELIZATION AND INFILL

After computing the intersection point using the Newton method described in Section 3, we perform the boundary voxelization and infill operation. We first identify the element's boundary voxels by comparing the intersection point with the individual voxel's location. We then perform an infill operation to assign the voxels between identified boundary voxels with the appropriate element index value using a marching operation. Finally, we identify the IGA evaluation points of the element in the vicinity of the voxel center and interpolate the IGA result value to assign to that voxel. The IGA evaluation points in an element can either be the element Gauss points or the element control points based on the availability of the IGA results.

### 4.1 Element Infill

We create an empty 3D voxel grid around the model based on the user-defined voxelization resolution and use the intersection points to identify the boundary voxels. We then use the  $t$  parameter obtained from the intersection to perform the infill operation, which obviates the need to store and sort the intersection data. Moreover, if there are duplicate intersection points from the same element, the voxel would be overwritten with the same element value, allowing for parallel processing without race conditions. However, some voxels on the element boundary might contain intersection points from multiple elements (see Figure 6). Parallelizing

the intersection test among the elements might result in a race condition, where multiple threads might write different element index values at the same voxel memory location. To avoid this race condition, we serialize the voxelization operation using an element `for` loop inside the GPU kernel. This will ensure that multiple intersections within a single voxel to be dealt with serially. After identifying all the intersection points on the element faces, we get the boundary voxels for the complete element (Figure 1(h)). We then perform a marching operation along the ray to identify the empty voxels between two boundary voxels (entry and exit voxel) and assign them the same element index value. Figure 6 shows a slice of voxel grid before and after the infill operation with the element values.



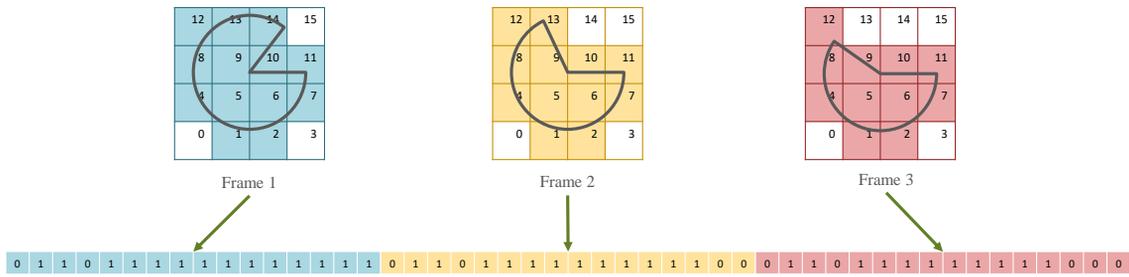
**Figure 6:** A ray might intersect the element boundary and could have multiple intersection points within the same voxel. In this example, the right voxel will need to store the intersection with both elements 1 and 2. A slice of the voxel grid after the infill operation is shown on the right.

## 4.2 IGA Result Assignment

The infilled model with the element values is further processed to get the variable field model with the IGA results. Each volume element has the IGA field results computed and stored either in the Gauss points or the control points. We have an element index value assigned to each voxel; we identify the IGA results field points associated with this element. The IGA field values are interpolated at the voxel center using Shepard's inverse distance weighting, as shown in Equation 5. Here  $k$  is the value of IGA result at  $i^{th}$  point and  $d(x, x_i)$  is the distance between one of the IGA result points and voxel center, and  $p$  is a positive real number called power parameter, usually 2. A higher value of  $p$  indicates a stronger influence of closer points.  $N$  is the number of IGA results points for each element.

$$k(x) = \begin{cases} \frac{\sum_{i=1}^N w_i(x) k_i}{\sum_{i=1}^N w_i(x)}, & \text{if } d(x, x_i) \neq 0, \text{ where } w_i = \frac{1}{d(x, x_i)^p} \\ k_i, & \text{if } d(x, x_i) = 0 \end{cases} \quad (5)$$

Shepard's equation is not an ideal way to assign IGA value to the voxel center in the case of skewed elements or elements with a high number of control points. In such cases, we use the closest point approach to compute the closest control point to the voxel center and assign its value. This approach is faster and gives visually accurate results for high-resolution models.



**Figure 7:** Individual voxelized model frame data saved using a flat data structure in GPU memory to use for animated volume rendering.

The values assigned to the voxel center are normalized to create a variable density model. This normalization allows for easy scaling and visualization using previously developed volume rendering tools [33]. Results of the voxelized IGA results are shown in Section 6.

## 5 ANIMATED VOLUME RENDERING

In this section, we outline the process for obtaining an animated volume rendering of the voxelized IGA model. The IGA results for each time frame are already stored on the GPU from previous operations. A screen pixel resolution ray casting algorithm with user-controlled sampling density is used to render each simulation time frame in the correct order.

### 5.1 Data Processing

The 3D voxel with the IGA result for each frame is converted into a flattened array. Figure 7 shows flattened array for the different time frames. This method facilitates a linear map between the time frames; the voxel resolution is used to set the GPU memory stride. The algorithm takes only a few milliseconds to process each frame and render it. Hence we can achieve real-time animated rendering of the frames. We can interact directly with the model by manipulating its view orientation and zoom. We can also dynamically section the model in a particular direction while being animated.

### 5.2 GPU Ray Casting

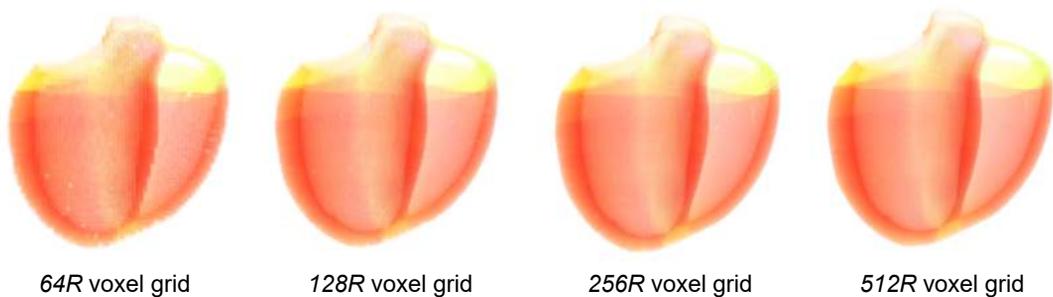
We perform the volume rendering of the voxelized model by implementing a ray-casting algorithm on the GPU, where each pixel on the screen corresponds to a single ray (pixel resolution rendering). We perform the sampling using a user-defined pitch to identify the voxels of the model that are along the selected ray. The total number of voxels sampled and the value of the density stored in each voxel are used to average the pixel color of the rendering. The ray casting algorithm comprises four steps: box intersection, sampling, shading, and compositing. We use a previously developed GPU ray casting method [33] to perform the animated volume rendering. We first compute the entry voxel of the ray by performing a ray box intersection with the complete bounding box of the voxel grid. We then march along the ray, sampling density values along the voxel grid. A critical parameter that determines the performance of the animated volume rendering is the sampling pitch. We need to select a finer pitch for high-resolution models to capture all the features along a ray, which affects the frame rates while rendering. However, even with high resolution models, we achieve interactive frame rates (see Section 6.3).

## 6 RESULTS

In this work, we present two IGA models to demonstrate our post-processing approach. The first is a 2-chamber cardiac model with stress and strain tensor values as IGA results at the Gauss points. The second is blood flow through an Aortic arch with a tricuspid valve. This is a fluid structure interaction (FSI) model with velocity and pressure values computed as IGA results. Both models have different data structures and require different interpolation methods for voxelization.

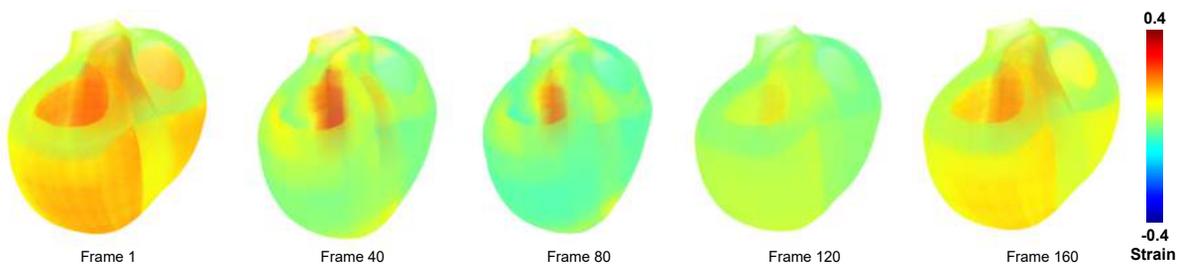
### 6.1 Cardiac Biomechanics Simulations

The cardiac model is from a biomechanics simulation study [14]. Data from this simulation is in the form of cubic-Hermite elements, which we convert to Bézier elements as explained in Figure 13. After conversion, the model consists of 162 cubic Bézier volumetric elements. Each volume element is decomposed into six Bézier surface patches, producing 972 surface patches. The complete simulation of a single heartbeat consists of 200 time steps.

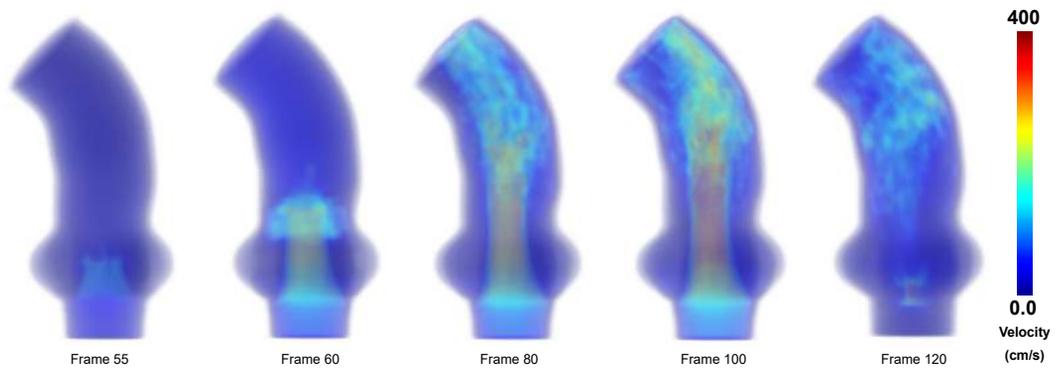


**Figure 8:** Volume rendering of the cardiac model at different resolutions.

The IGA results contain stress and strain tensor values for the individual volumetric elements. We perform Shepard's interpolation during results assignment to individual voxels as described in Section 4.2. The volume rendering of the cardiac model with different voxel resolution are shown in Figure 8. Figure 9 represents the first invariant of the strain values assigned to the voxelized model with a resolution of 256R at different time frames (see Table 1 for details on the voxel grid). The timing data of various cardiac model resolutions are presented in Section 6.



**Figure 9:** Animated volume rendering of the cardiac model showing the first invariant of the strains at different time frames at 512R resolution.



**Figure 10:** Animated volume rendering of the Aorta model showing the fluid velocity at different time frames.

## 6.2 Aortic FSI Simulation

The Aortic model is from a fluid-structure interaction simulation study of blood flow in a flexible aorta with an immersed aortic valve [32, 31]. The model consists of 27 volumetric NURBS elements. Each of the six NURBS faces of each element is decomposed into Bézier patches of degree 2 using the knot insertion algorithm [25]. After decomposition, model is consists of 38070 Bézier patches (see Figure 2). The analysis consists of 161 time frames for the complete cardiac cycle.

In this model, the IGA results are computed at the control points. Each frame contains 145962 control points with pressure and velocity values associated with them. The voxels corresponding to the elements are first identified, and then the value of the field at the closest control point to the voxel center is used for the IGA result assignment. Figure 10 shows the magnitude of the velocity in a model with a voxel resolution of  $256R$  at different time frames.

## 6.3 Timing Results

In this section, we present timing results for both the cardiac model and the aorta model. We present timings for two approaches for processing the results. They are batch processing, where we process all the timesteps in parallel and sequential processing, where we process each time step sequentially. Since there is a strict upper limit on the GPU memory, we could not perform the batch processing of all the frames at the highest resolution. However we performed sequential processing at four different resolutions:  $64R$ ,  $128R$ ,  $256R$ ,  $512R$  (see Table 1). The testing was performed on one node of an HPC cluster, with Intel Xeon 6140 processors and NVIDIA Tesla V100 GPU with 32GB memory. The voxelized output files are stored in a binary format and visualized using the animated volume rendering on the GPU. We also recorded the frame rates achieved while

**Table 1:** Grid resolutions for voxels in 3D for both test models.

Resolution	Heart	Aorta
64R	64 × 61 × 49	36 × 31 × 64
128R	128 × 121 × 97	71 × 61 × 128
256R	256 × 242 × 192	140 × 121 × 256
512R	512 × 482 × 383	279 × 240 × 512

**Table 2:** Memory requirements for sequential and batch processing of the cardiac and aorta model.

Resolution	Cardiac Model (GB)		Aorta Model(GB)	
	Batch	Sequential	Batch	Sequential
64R	0.464	0.407	1.950	2.501
128R	1.686	0.417	3.003	2.505
256R	11.364	0.495	6.403	2.528
512R	-	1.112	-	2.694

**Table 3:** Timing data for the cardiac model.

Resolution	Batch Processing Time (s)				Sequential Processing Time (s)				Frame Rate (fps)
	Memory Allocation	Ray Intersection	File Save	Total Time	Memory Allocation	Ray Intersection	File Save	Total Time	
64R	0.102	0.786	0.584	1.472	0.083	19.609	0.798	20.657	39.9
128R	0.360	3.558	2.510	6.440	0.084	26.803	2.560	29.848	36.7
256R	2.342	20.873	21.788	45.046	0.102	90.126	22.052	114.341	49.9
512R	-	-	-	-	0.212	543.151	165.582	717.709	49.6

**Table 4:** Timing data on aorta model.

Resolution	Batch Processing Time (s)				Sequential Processing Time (s)				Frame Rate (fps)
	Memory Allocation	Ray Intersection	File Save	Total Time	Memory Allocation	Ray Intersection	File Save	Total Time	
64R	1.055	2.660	0.829	4.544	1.114	11.354	0.887	13.354	35.5
128R	1.137	18.172	3.956	23.266	1.275	29.220	3.851	34.346	35.3
256R	1.827	133.673	30.509	166.009	1.941	146.463	30.314	178.719	36.1
512R	-	-	-	-	4.718	1001.775	279.583	1286.056	38.5

rendering the animated frame. Except for reading the data files (see [Figure 1](#)), all operations are performed on the GPU. This approach reduces data transfer between CPU and GPU, which significantly improves the performance of our approach.

We could not test the 512R resolution for the cardiac model with batch processing due to GPU memory limitations. [Table 3](#) shows timing data for the batch and sequential processing for different resolutions of the cardiac model. The data shows that sequential processing can be performed for higher resolutions since the peak memory utilization is lower (see Memory table). Batch frame processing is more efficient, but it comes at a considerable memory cost compared to sequential frame processing.

Similarly, for the aorta model, the 512R resolution could not be tested for batch processing due to GPU memory utilization. The memory utilization is also overall higher for the aorta model due to the higher number of patches (38070 bi-quadratic Bézier patches and 145962 control points with IGA results). [Table 4](#) shows the timing results for the sequential and batch processing of the aorta model results. Overall, the aorta model processing is slower than that of the cardiac model since the aorta model has a relatively higher number of

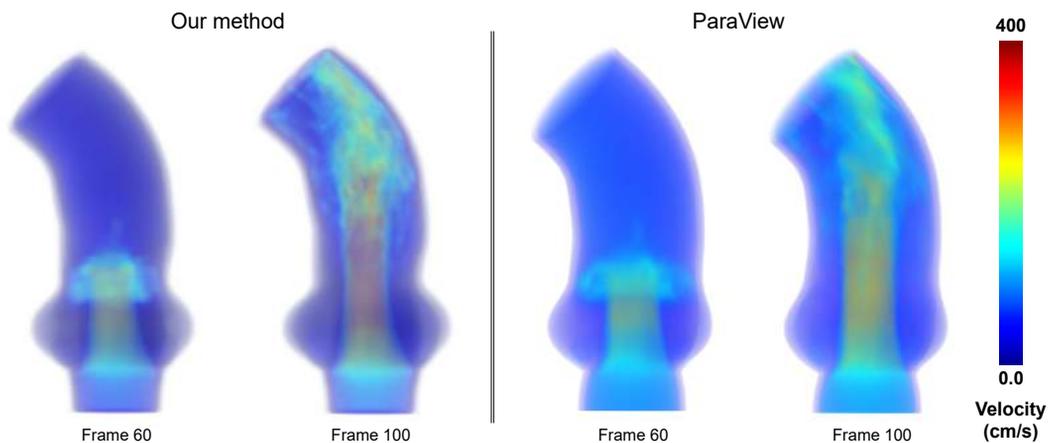
patches. The batch process is significantly faster than sequential processing; however, we cannot process higher resolutions due to the GPU memory constraint.

For each resolution, the voxel models were computed for 200 frames of the cardiac simulation and 161 frames for the Aorta FSI simulation. Once the frames are loaded into the GPU memory, they are rendered directly using the frame number and animated interactively. The frame rates were assessed while animating and rendering the voxelized model from a static point of view in full HD screen resolution ( $1920 \times 1080$ ) using GPU ray casting on an NVIDIA Titan Xp GPU. As seen in the frame rate column of Table 3 and Table 4, the frame rates were consistently above 30 fps for all resolutions. For the  $256R$  and  $512R$  voxelized model, the frame stride was more than one due to GPU memory limitations. This higher stride results in a higher frame rate for  $256R$  and  $512R$  compared to the lower resolution models.

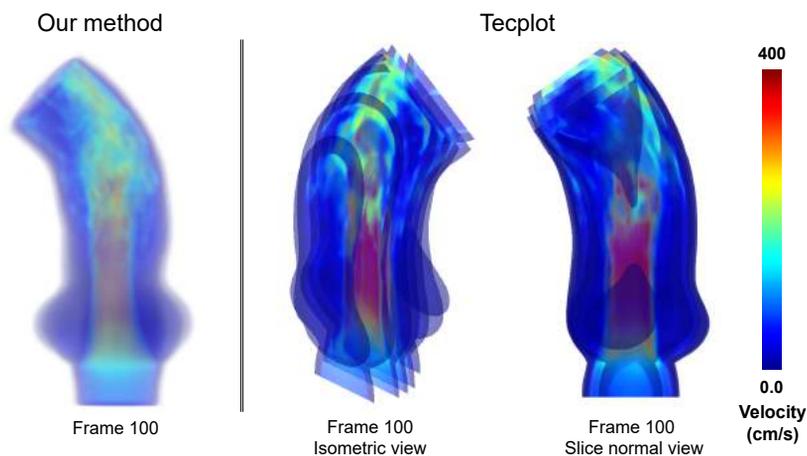
The supplemental material provided with this paper includes a video demonstration for both models. Animations presented in the video are at  $256R$  resolution for both models. We display dynamic sectioning of the model while animating the results. We can also change the orientation (zoom, rotation) of the model to inspect the analysis results in detail.

#### 6.4 Comparison with Commercial Visualization Tools

We compared our volume rendering approach with commercial visualization tools, Tecplot and ParaView. In the case of Tecplot, the user has to generate multiple slices and stack them over each other with transparency to visualize internal volumetric results (see Figure 12). In ParaView, the user can load the entire model and convert it to a volumetric model, but the rendering is slow depending on the complexity of the model (Figure 11). We tested the Aorta model in both these tools to compare them against our method. Our comparison shows that our framework is significantly better in terms of the preprocessing and rendering time. Moreover, the rendering generated from our approach allows the user to interact with the model (change view orientation, zoom, slice) dynamically while animating the analysis results. Our interaction with the model can also involve changing the visualization colormap and the model density during the animation.



**Figure 11:** Selected frames for the Aorta model generated using our approach and the same frames generated using ParaView.



**Figure 12:** Comparison of our method with Tecplot. Since Tecplot does not have inherent volume rendering capabilities, it has to be simulated by stacking multiple sections of the model with transparency.

## 7 CONCLUSIONS

We have presented a framework to render the results of dynamic isogeometric analysis simulations. This approach uses ray intersections to convert volumetric splines to a time-varying voxel representation. We have developed two techniques, sequential and batch frame processing. Sequential processing of frames overcomes the GPU memory limit and enables voxelization at a higher resolution. The time-varying voxel models are then volume rendered using a GPU-accelerated ray casting method to animate any result fields from the IGA simulations. The animation can be sectioned to study the quantities of interest in greater detail without compromising the interactive frame rates. The GPU-accelerated animated volume rendering method can achieve a frame rate of over 30 fps at a full HD resolution. Consequently, this allows interactive interrogation of the IGA results that can yield valuable insights into the physical simulation.

## ACKNOWLEDGEMENTS

We would like to thank Dr. Hsu at Iowa State University for providing the Aorta model and IGA data for the FSI simulation. This work was partially supported by the National Institutes of Health under award number R01HL131753 and the National Science Foundation under grant OAC-1750865. We would also like to thank NVIDIA® for providing GPUs used for testing the algorithm developed during this research.

Adarsh Krishnamurthy, <http://orcid.org/0000-0002-5900-1863>

## REFERENCES

- [1] Abert, O.; Geimer, M.; Müller, S.: Direct and fast ray tracing of NURBS surfaces. RT'06: IEEE Symposium on Interactive Ray Tracing 2006, Proceedings, 161–168, 2007. <http://doi.org/10.1109/RT.2006.280227>.
- [2] Cignoni, P.; De Floriani, L.; Montani, C.; Puppo, E.; Scopigno, R.: Multiresolution modeling and visualization of volume data based on simplicial complexes. In Proceedings of the 1994 Symposium on Volume Visualization, 19–26, 1994.

- [3] Costa, K.; Hunter, P.; Wayne, J.; Waldman, L.; Guccione, J.; McCulloch, A.: A three-dimensional finite element method for large elastic deformations of ventricular myocardium: II-prolate spheroidal coordinates. *Journal of Biomechanical Engineering*, 118(4), 464–472, 1996.
- [4] Cottrell, J.A.; Hughes, T.J.R.; Bazilevs, Y.: *Isogeometric Analysis: Toward Integration of CAD and FEA*. John Wiley & Sons, Chichester, 2009.
- [5] Cottrell, J.A.; Hughes, T.J.R.; Reali, A.: Studies of refinement and continuity in isogeometric structural analysis. *Computer Methods in Applied Mechanics and Engineering*, 196, 4160–4183, 2007.
- [6] Geimer, M.; Abert, O.: Interactive ray tracing of trimmed bicubic Bézier surfaces without triangulation. *Winter School of Computer Graphics (WSCG 2005)*, 71–78, 2005. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.123.823>.
- [7] Hsu, M.C.; Wang, C.; Herrema, A.J.; Schillinger, D.; Ghoshal, A.; Bazilevs, Y.: An interactive geometry modeling and parametric design platform for isogeometric analysis. *Computers and Mathematics with Applications*, 70(7), 1481–1500, 2015.
- [8] Hughes, T.J.R.; Cottrell, J.A.; Bazilevs, Y.: Isogeometric analysis: CAD, finite elements, NURBS, exact geometry, and mesh refinement. *Computer Methods in Applied Mechanics and Engineering*, 194, 4135–4195, 2005.
- [9] Intersection: Line-plane intersection. Wikipedia, 2018–. [https://en.wikipedia.org/wiki/Line-plane\\_intersection](https://en.wikipedia.org/wiki/Line-plane_intersection). [Online; accessed May 2018].
- [10] Kajiya, J.T.: Ray Tracing parametric patches. *Computer Graphics*, 16, 245–254, 1982.
- [11] Kajiya, J.T.; Von Herzen, B.P.: Ray Tracing Volume Densities. *SIGGRAPH Comput. Graph.*, 18(3), 165–174, 1984. ISSN 0097-8930. <http://doi.org/10.1145/964965.808594>.
- [12] Knoll, A.; Wald, I.; Parker, S.; Hansen, C.: Interactive isosurface ray tracing of large octree volumes. In *2006 IEEE Symposium on Interactive Ray Tracing*, 115–124, 2006.
- [13] Kong, H.: A new method for speeding up ray tracing NURBS surfaces. *Computers and Graphics (Pergamon)*, 21(5), 577–586, 1997.
- [14] Krishnamurthy, A.; Gonzales, M.J.; Sturgeon, G.; Segars, W.P.; McCulloch, A.D.: Biomechanics simulations using cubic Hermite meshes with extraordinary nodes for isogeometric cardiac modeling. *Computer Aided Geometric Design*, 43, 27–38, 2016. ISSN 0167-8396. <http://doi.org/10.1016/J.CAGD.2016.02.016>.
- [15] Kumar, S.; Manocha, D.: Efficient rendering of trimmed nurbs surfaces. *Computer-Aided Design*, 27(7), 509–521, 1995. ISSN 0010-4485. [http://doi.org/10.1016/0010-4485\(94\)00003-V](http://doi.org/10.1016/0010-4485(94)00003-V).
- [16] Laurent, B.: Implicit matrix representations of rational Bézier curves and surfaces. *CAD Computer Aided Design*, 46(1), 14–24, 2014. ISSN 00104485. <http://doi.org/10.1016/j.cad.2013.08.014>.
- [17] Levoy, M.: Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3), 245–261, 1990.
- [18] Lorensen, W.E.; Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4), 163–169, 1987.
- [19] Martin, T.; Cohen, E.; Kirby, R.M.: Direct isosurface visualization of hex-based high-order geometry and attribute representations. *IEEE Transactions on Visualization and Computer Graphics*, 18(5), 753–766, 2012.
- [20] Martin, W.; Cohen, E.: Representation and extraction of volumetric attributes using trivariate splines: A mathematical framework. In *Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications*, 234–240, 2001.
- [21] Martin, W.; Cohen, E.; Fish, R.; Shirley, P.: Practical Ray Tracing of Trimmed NURBS Surfaces. *J. Graph. Tools*, 5(1), 27–52, 2000. ISSN 1086-7651. <http://doi.org/10.1080/10867651.2000.10487519>.

- [22] Nelson, B.; Kirby, R.M.: Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(1), 114–125, 2006.
- [23] Owen, S.G.: Ray-Box intersection, 1999. <https://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter0.htm>.
- [24] Pabst, H.F.; Springer, J.P.; Schollmeyer, A.; Lenhardt, R.; Lessig, C.; Froehlich, B.: Ray casting of trimmed NURBS surfaces on the GPU. *RT'06: IEEE Symposium on Interactive Ray Tracing 2006, Proceedings*, 151–160, 2007. <http://doi.org/10.1109/RT.2006.280226>.
- [25] Piegl, L.; Tiller, W.: *The NURBS Book*. Springer, 1997. ISBN 9783540615453.
- [26] Purcell, T.J.; Buck, I.; Mark, W.R.; Hanrahan, P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3), 2002. ISSN 07300301. <http://doi.org/10.1145/566654.566640>.
- [27] Ray, H.; Pfister, H.; Silver, D.; Cook, T.A.: Ray casting architectures for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3), 210–223, 1999. ISSN 10772626. <http://doi.org/10.1109/2945.795213>.
- [28] Rushmeier, H.; Hamins, A.: Volume rendering of pool fire data. *IEEE Computer Graphics and Applications*, 15(4), 62–67, 1995. ISSN 0272-1716. <http://doi.org/10.1109/38.391493>.
- [29] Shen, J.; Busé, L.; Alliez, P.; Dodgson, N.: A line/trimmed NURBS surface intersection algorithm using matrix representations. *Computer Aided Geometric Design*, 48, 1–16, 2016. ISSN 01678396. <http://doi.org/10.1016/j.cagd.2016.07.002>.
- [30] Toth, D.: On Ray Tracing Parametric Surfaces. *ACM SIGGRAPH Computer Graphics*, 19(3), 171–179, 1985. ISSN 0097-8930. <http://doi.org/10.1145/325334.325233>.
- [31] Wu, M.C.; Zakerzadeh, R.; Kamensky, D.; Kiendl, J.; Sacks, M.S.; Hsu, M.C.: An anisotropic constitutive model for immersogeometric fluid–structure interaction analysis of bioprosthetic heart valves. *Journal of Biomechanics*, 74, 23–31, 2018.
- [32] Xu, F.; Morganti, S.; Zakerzadeh, R.; Kamensky, D.; Auricchio, F.; Reali, A.; Hughes, T.J.; Sacks, M.S.; Hsu, M.C.: A framework for designing patient-specific bioprosthetic heart valves using immersogeometric fluid–structure interaction analysis. *International journal for numerical methods in biomedical engineering*, 34(4), e2938, 2018.
- [33] Young, G.; Krishnamurthy, A.: GPU-accelerated generation and rendering of multi-level voxel representations of solid models. *Computers and Graphics (Pergamon)*, 75, 11–24, 2018. ISSN 00978493. <http://doi.org/10.1016/j.cag.2018.07.003>.
- [34] Zhang, Q.; Eagleson, R.; Peters, T.M.: Volume visualization: A technical overview with a focus on medical applications. *Journal of Digital Imaging*, 24(4), 640–664, 2011. ISSN 08971889. <http://doi.org/10.1007/s10278-010-9321-6>.

### Hermite to Bézier Conversion

In Hermite elements, the nodes contain information about its value as well as the derivatives. This data has a one-to-one mapping to a Bézier element. The control point locations for the Bézier elements can be computed using Equation 6. These equations are specifically formulated to calculate the  $4 \times 4$  control points for a cubic Bézier element.  $P$  are control points.  $P_{00}$  is one of the corner point of the Hermite patch, which is used to compute 3 control points:  $P_{10}, P_{01}, P_{11}$ . This process is repeated to obtain the other 9 control points from the remaining 3 corner points. Selection of + or - is based on the relative location of the control point with respect to the corner control point used for computation as shown in Figure 13.

$$\begin{aligned}
 P_{10} &= P_{00} \pm \frac{1}{3} \left( \frac{dP}{du} \right)_{(0,0)} \\
 P_{01} &= P_{00} \pm \frac{1}{3} \left( \frac{dP}{dv} \right)_{(0,0)} \\
 P_{11} &= P_{00} \pm \frac{1}{3} \left( \frac{dP}{du} \right)_{(0,0)} \pm \frac{1}{3} \left( \frac{dP}{dv} \right)_{(0,0)} \pm \frac{1}{9} \left( \frac{d^2P}{dudv} \right)_{(0,0)}
 \end{aligned}
 \tag{6}$$

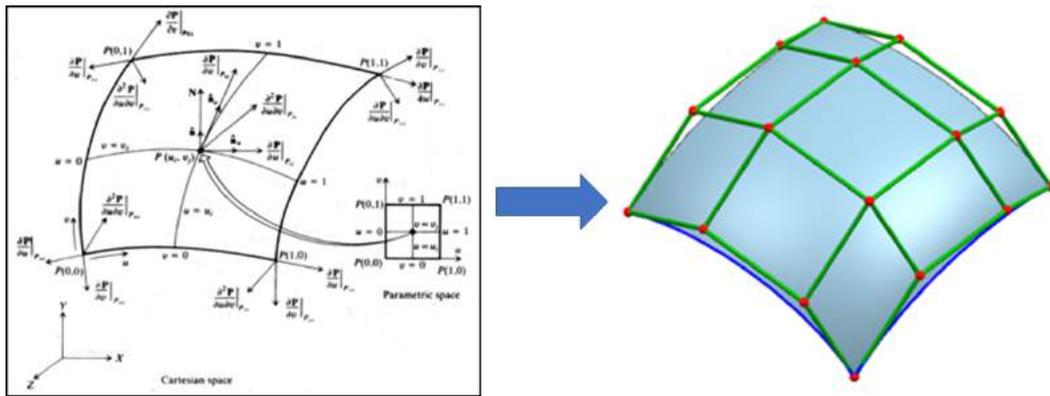


Figure 13: Conversion of Hermite surface patches into Bézier control mesh.

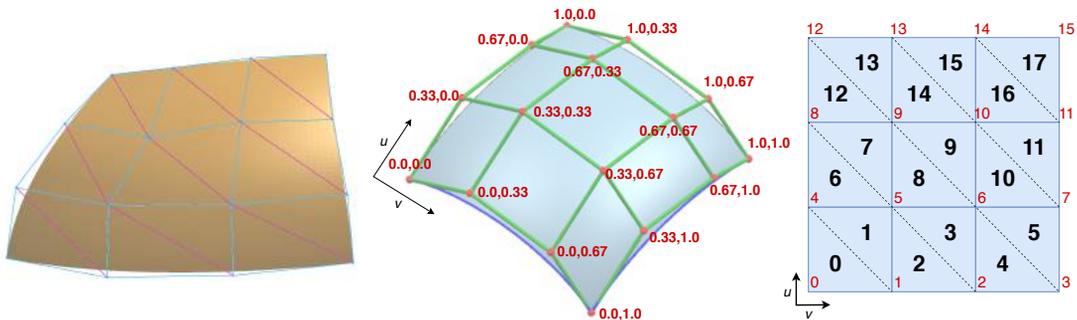


Figure 14: Example triangulation of Bézier element control point grid (left), knot intervals assigned to control points grid of  $4 \times 4$  (center), and the triangulation of a  $4 \times 4$  control point grid (right).