Accelerating Unstructured Mesh Point Location with RT Cores

Nate Morrical[‡] Ingo Wald[†] Will Usher[‡] Valerio Pascucci[‡]
[‡]SCI Institute, University of Utah [†]NVIDIA

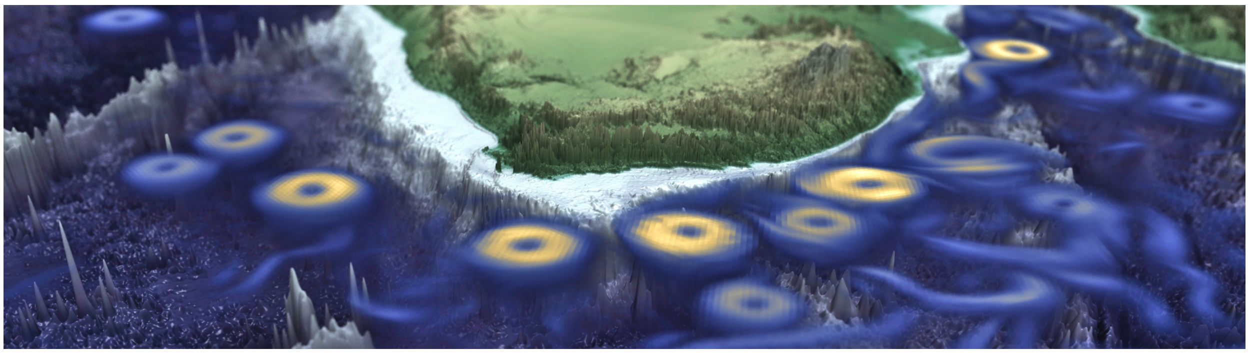


Fig. 1: *The Agulhas Current dataset, courtesy Niklas Röber, DKRZ. This image shows simulated ocean currents off the coast of South Africa, represented using cell-centered wedges. When rendered using our hardware accelerated point queries, we see up to a 14.86× performance improvement over a CUDA reference implementation (2.49 FPS vs 37 FPS on an RTX 2080 at* 1024×1024*).*

Abstract—We present a technique that leverages ray tracing hardware available in recent Nvidia RTX GPUs to solve a problem other than classical ray tracing. Specifically, we demonstrate how to use these units to accelerate the point location of general unstructured elements consisting of both planar and bilinear faces. This unstructured mesh point location problem has previously been challenging to accelerate on GPU architectures; yet, the performance of these queries is crucial to many unstructured volume rendering and compute applications. Starting with a CUDA reference method, we describe and evaluate three approaches that reformulate these point queries to incrementally map algorithmic complexity to these new hardware ray tracing units. Each variant replaces the simpler problem of point queries with a more complex one of ray queries. Initial variants exploit ray tracing cores for accelerated BVH traversal, and subsequent variants use ray-triangle intersections and per-face metadata to detect point-in-element intersections. Although these later variants are more algorithmically complex, they are significantly faster than the reference method thanks to hardware acceleration. Using our approach, we improve the performance of an unstructured volume renderer by up to $4\times$ for tetrahedral meshes and up to $15\times$ for general bilinear element meshes, matching, or out-performing state-of-the-art solutions while simultaneously improving on robustness and ease-of-implementation.

Index Terms—Scientific Ray Tracing, Unstructured Scalar Data, GPGPU, Simulation, Volume Rendering

1 Introduction

Even before the first programmable GPUs, researchers have been finding new ways to cleverly reformulate their algorithms to take advantage of specialized graphics hardware [25]. These graphics accelerators started as relatively simple devices that offloaded only certain parts of the rasterization pipeline, but have since evolved into massively parallel processors with a wide range of applications. As hardware has progressed, it is easier, now more than ever, to use these GPUs for general computation. However, GPUs still contain a significant amount of dedicated hardware resources that offer the potential to accelerate workloads beyond what current frameworks enable, and have yet to be explored to their fullest.

Our work focuses on exploring the ray tracing (RT) cores new to Nvidia's Turing architecture (via the "RTX" platform), which can be used to achieve compelling visual effects like reflections and refractions, soft shadows, and global illumination [4]. Algorithms like ray tracing heavily involve tree traversal to locate and test intersections of rays against primitives, and have traditionally been difficult to parallelize on GPU architectures, as summarized by Vinkler et al [37]. Tree traversal tends to be inherently divergent, resulting in a reduction of parallelism, instruction cache thrashing, and many incoherent reads to memory–all of which significantly degrade GPU performance. Once candidate leaves are found, the large number of primitive intersection tests required can also be prohibitively expensive. These RT cores help accelerate this process by performing bounding volume hierarchy (BVH) traversal and ray-triangle intersections in hardware [9], freeing up existing GPU resources to focus on shading computation.

Beyond ray tracing, we believe these RT cores can be used for general purpose computation. In geometry processing, BVH traversal is essential for closest point queries [33]. In simulation, both BVH traversal as well as primitive intersection testing are used for collision detection [10], for mesh contact deformations [42], and for adjacency queries [24]. And in visualization, these operations are required for sample reconstruction of unstructured meshes during volumetric rendering [28]. If carefully reformulated into a "ray tracing" problem, applications like these could likely leverage these RT cores as well.

This paper explores a proof-of-concept that leverages these RT cores to solve a problem other than classical ray tracing. Specifically, we develop a technique that uses these RT cores for volumetric rendering of

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

large unstructured meshes through the use of point queries. This point query method is particularly attractive for unstructured mesh volume rendering, as it integrates nicely with existing regular grid methods like adaptive ray marching, empty space skipping, and stochastic path tracing. Traditional point query methods used to render large unstructured volumes require clusters of CPU nodes to achieve interactive frame rates [2, 28]. However, these clusters are inaccessible to many in the visualization community, and performance is limited when constrained to a single workstation. Other prior works [39] make the assumption that these unstructured volumes contain only tetrahedral elements. However, many unstructured data sets contain a mix of both tetrahedral as well as higher dimensional elements like pyramids, wedges, and hexes [1].

We show that these data sets can be visualized interactively on a single GPU workstation by reformulating the process of point location to use ray tracing hardware. We additionally show that it is possible to extend the use of these RT cores to support more general, nonlinear unstructured elements, despite the presence of non-triangular geometry. The methods we present progressively build off of each other to incrementally map the algorithmic complexity of these point location queries to different aspects of the ray tracing hardware. Finally, we evaluate our solution on a mix of synthetic and real world applications, and show that our approach matches or outperforms state-of-the-art while simultaneously reducing implementation complexity by relying on hardware to perform otherwise involved operations.

2 Related Works

There are two separate collections of prior works that both relate to our work, although in different ways. First, our work demonstrates the use of RT cores for General Purpose GPU (GPGPU) Computation. There is an interesting history to GPGPU strategies that we draw inspiration from, and so we briefly summarize the body of work in this area in Section 2.1. Following that, in Section 2.2, we cover the prior works of our targeted example use case—volume visualization of large, mixed element unstructured meshes.

2.1 General Purpose GPU Computation

Many previous applications have successfully shown that it is possible to leverage specialized GPU hardware for general computation [25].

2.1.1 Fixed-Function Hardware

Before the development of programmable shaders, researchers were already demonstrating how GPUs could be used to accelerate several general applications. One of the earliest of these techniques was presented by Larsen and McAllister [14], who accelerated large matrix-matrix multiplication by actually "visualizing" the matrix computation with graphics hardware. Moreland and Angel [16] successfully implemented a fast Fourier Transform which took advantage of bitmap and frame buffer operations. Rumpf and Strzodka [30, 31] demonstrated the potential of register combiners for accelerating Finite-Element simulations, as well as for computing level sets of regular-grid image data.

2.1.2 Programmable Shaders

As hardware evolved, programmable vertex and fragment shaders were introduced, which allowed users to perform custom per-vertex and per-fragment operations in parallel. Many researchers investigated the use of these shaders in accelerating physically based simulations. Green [5] demonstrated the use of these shaders in accelerating cloth simulation. Kim and Lin [11] used these shaders to implement a partial-differential-equation solver to model the growth of ice crystals. Krüger and Westermann [13] described a technique for simulating volumetric effects by rendering iterations into 2D textures using fragment shaders.

2.1.3 Post-CUDA Fixed-Function Hardware

Since the introduction of CUDA [21] and OpenCL [35] in the early 2000s, research in exploiting graphics hardware has calmed down a bit. These frameworks enabled developers to leverage many GPU capabilities directly in a general-purpose programming language. Still, specialized hardware units have been introduced since then that can be leveraged for general computation.

Tensor Cores. With the Volta architecture came tensor cores, which accelerate large tensor multiply and accumulate operations for use in accelerating AI and machine learning applications. Although not the focus of this work, tensor cores have the potential to accelerate many general purpose applications as well. Haidar et al. [8] were able to apply these cores to iterative linear system solvers, extending the hardware's use to many general scientific computing problems.

Ray-Tracing Cores. Most relevant to our work are the ray tracing (RT) cores introduced with Turing. Traditionally, acceleration structure traversal has been a difficult task to optimize for GPU architectures, as summarized by Vinkler et al [37]. However, with these RT cores, bounding volume hierarchy (BVH) traversal and ray-triangle intersections are now accelerated in hardware. Since the publication of our short paper [39], several works have leveraged these cores for other tasks beyond ray tracing for rendering. Ganter et al. [3] demonstrated that ray tracing cores could be used to more efficiently skip empty space in the context of a structured data ray caster. Morrical et al. [17] also showed that ray tracing cores could be used to skip empty space, but in the context of unstructured data instead of regular grid data. Their work further demonstrated an adaptive sampling scheme that used ray tracing cores to fetch metadata about a local region in space for use in adaptive volume sampling. Knoll et al. [12] demonstrated that these cores could be used to accelerate particle sorting in the context of an efficient particle volume splatter. Wald et al. [41] make use of ray tracing cores for adaptive mesh refinement visualization. In the simulation domain, Salmon et al. [32] leveraged ray tracing cores to accelerate a Monte Carlo particle transport simulation code. Ulmstedt and Joacim [36] use ray tracing cores to simulate the propagation paths of sound in water. Zellmann et al. [43] proposed a technique that uses ray tracing cores to simulate force directed graphs.

2.2 Unstructured Volume Rendering

Unstructured volume visualization is a challenging problem in the scientific visualization community. In the context of unstructured volume rendering, the bulk of existing methods can only visualize an approximation of the true underlying unstructured volumetric data, and suffer from algorithmic complexity issues as data sets grow larger. Many of these algorithms have focused on rasterization based GPU methods. The early work by Shirley and Tuchman [34] approximate direct scalar volume rendering of unstructured meshes by sorting and rasterizing a collection of tetrahedra from front to back each frame. Maximo et al. [15] use CUDA to accelerate this sorting process, and use programmable shaders to avoid multiple draw calls. However, this sort is still prohibitively expensive. At the time, Maximo et al. were only able to reorder 6 million tetrahedra per second, while we demonstrate interactive rendering on datasets up to an order of magnitude larger.

More recent works tend to prefer ray-casting approaches to volume rendering rather than rasterization techniques, as ray marching does not require reordering the data during camera movement, and can be terminated early if a pixel reaches maximum opacity. One such way to do this ray-casting process on unstructured volumes, as demonstrated by Muigg et al. [18], is to compute a set of per-pixel face-sequence lists that can be used to march the ray from one element to the next. Although this technique does improve performance over rasterization

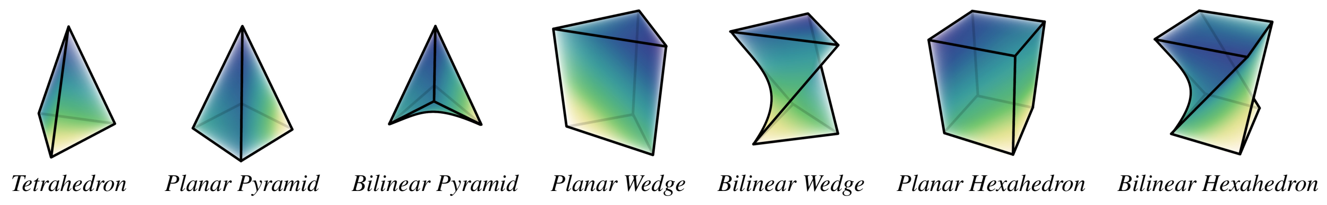


Fig. 2: *In this work, we'll evaluate different ways to leverage ray tracing hardware to accelerate point location within the above elements for use in volume rendering. Given an unstructured mesh composed of the above elements as well as an arbitrary 3D point, determine which element that point is in and return it to the user. Each element can be composed of entirely triangular faces, or of more complex configurations of triangles, planar quads, and bilinear patches.*

based alternatives, the overhead of traversing these face-sequence lists limits performance improvements when compared to a CPU reference. This traversal from cell to neighboring cell tends to dominate rendering performance, especially as unstructured meshes become more dense. The work by Gu and Kim [7] attempt to correct some of the accuracy and performance limitations of the work by Muigg et al. Their work avoids construction of linked-lists on the GPU by instead using more memory-coherent arrays. Although their technique improves performance over Muigg, their approach remains non-interactive for large meshes, taking four to five seconds to render a 41 million tetrahedral mesh on a Pascal GPU.

Another approach to unstructured volume visualization, as demonstrated by Childs et al. [2], is to rasterize the unstructured elements into a regular grid. Once rasterized, texture units can be used to efficiently query elements at a given set of coordinates during ray marching. These texture units work exceptionally well when neighboring threads query from similar locations; however, rasterized versions of unstructured data can quickly become too expensive to store in memory and on disk. To reduce memory usage the grid can be coarsened; however, if made *too coarse*, the rasterization process may lose important details in the data. In these cases, rasterization may be infeasible to do, or produce a poor-quality representation that is unsuitable for use.

Outside of GPU unstructured mesh visualization, the recent work by Rathke et al [28] relies heavily on tree traversal for direct sample reconstruction and level-set extraction of unstructured volume data. By sampling the unstructured elements directly, Rathke et al. can render unstructured volumes in a nearly identical process to traditional regular grid volume rendering, without any loss in accuracy or significant increase in memory usage. This has the advantage of enabling existing optimizations that have traditionally been limited to regular grid volume rendering, like adaptive sampling and empty space skipping [17]. Their approach, when run on a dual socket CPU workstation, is able to achieve semi-interactive frame rates that beat out many prior GPU methods as unstructured meshes grow larger.

3 RTX Beyond Ray Tracing

The overarching goal of this paper is to investigate and evaluate different ways to leverage ray tracing hardware to accelerate direct unstructured mesh point location. In the following sections, we will describe several different kernel iterations (also see Figure 1). First, we describe a reference CUDA kernel that traverses a BVH without any RT core acceleration whatsoever (Section 4). Next, we describe a kernel that uses the RT cores for BVH traversal (Section 5), followed by two different kernels (Section 6) that exploit both hardware BVH traversal and hardware ray-triangle intersection. New to this extension, we modify our final kernel to support general elements with potentially non-linear faces (Section 7) and explain any additional preprocessing required. Finally, we evaluate these kernels using artificial point query benchmarks and a proof-of-concept volume ray marcher for unstructured meshes on a broad range of commodity and high-end GPUs, both with and without hardware-accelerated ray tracing (Section 9).

3.1 Kernel Interface

All four kernels use the same interface: Given an unstructured mesh and an arbitrary 3D point, determine which element that point is in and return it to the user. For each kernel we look at two variants: one that returns just the ID of the element containing the point, and one that returns a scalar field value for the query point (either by interpolating a per-vertex scalar, or looking up a per-cell scalar, as provided by the data set). If the point is not contained in any element, the kernels return an ID of -1 or a scalar value of $-\infty$, respectively.

3.2 Input Element Types

The input to each of our kernels is an unstructured mesh that consists of an array of `float3` vertices and an array of eight `int` indices. During the point query, we can determine the element type from the number of non-negative indices. For tetrahedra, the fifth through eighth indices are -1. For pyramids, the sixth through eighth indices are -1. For wedges, the seventh and eighth indices are -1, and for hexahedra, all indices are greater than or equal to 0. Note that each element type that we demonstrate in Figure 2 requires a different number of indices, and for certain elements like tetrahedra, eight indices per element would be inefficient memory-wise. However, we use eight indices for all elements for simplicity. For the scalar field kernel, the data set also provides an additional `float` array of per-vertex or per-cell scalars.

The faces of these unstructured elements can be triangles or bilinear quads. Although the edges of these bilinear quads are always linear, since the corners of these quads might not necessarily be co-planar, the surface of these faces may curve quadratically to meet all four corners. Any point in the interior of the bilinear surface can be obtained by interpolation of a u and v between 0 and 1 using the following equation:

$$Q(u,v) = P_0(1-u)(1-v) + P_1(1-u)v + P_2u(1-v) + P_3uv \quad (1)$$

Note that bilinear quads are always order 2 surfaces, with order 1 interpolants along u and v, and have no inflection points. Although higher-order faces are possible for general unstructured data sets, for simplicity, we will be focusing on only bilinear elements.

3.3 Implementation

We implement all our kernels within OptiX [26], which added support for Turing's hardware accelerated ray tracing capabilities through Nvidia's RTX platform in version 6. We do assume basic familiarity with both OptiX and Nvidia's RTX platform, and refer the reader to the latest OptiX Programming Guide [22] and the Turing whitepaper [23] for reference. Beyond portability, one advantage of OptiX is that it uses CUDA under the hood, which allows us to evaluate our CUDA-only reference method within the same framework as our RTX optimized methods. We also make use of OptiX's template support to guarantee code consistency (i.e., point-in-element testing, the volume renderer's ray marching, transfer function lookup code, etc.) across our kernels.

```
int pointLocationReference(vec3f P)
  stack = { rootNode };
  while (!stack.empty())
    nodeRef = stack.pop();
    if (nodeRef is leaf)
    if (pointInElement(P, nodeRef.getChild()))
      return element
    else foreach child : 0..4
    if (pointInBox(P, nodeRef.getBounds(child))
      stack.push(nodeRef.getChildRef(child));
  return -1; /* no containing element */
```

Fig. 3: *Pseudocode for the reference method we'll be comparing against. This method performs a point query by traversing a four-wide BVH and performing point-in-element tests at the leaves.*

For the point-in-tetrahedra test we use the 3D version of Cramer's method (also known as Pineda's method [27]) to compute the four barycentric coordinates of p, and test if they are all non-negative. If all are positive, the four values can then, if desired, be used for interpolating the per-vertex scalar values. For all other general, *non-linear* elements, the scalar field interpolants cannot (to our knowledge) be inverted analytically. Instead, we use a root-finding algorithm—specifically Newton's method as done in OSPRay [38]—to determine if a point lies within an element, and if required, how to interpolate that element's per vertex values. These Newton-Raphson iterations become very efficient when the optimization is initialized to be close to the underlying solution where the iterations exhibit quadratic convergence.

4 Non-RTX Reference: cuda-bvh

To provide a non-RTX reference method, we first implemented a software-based unstructured-mesh point query in CUDA. Our implementation is similar to how such queries are done in OSPRay [38], using the method described by Rathke et al. [28]. Similar to Rathke et al., we build a BVH over the unstructured elements comprising the volume; however, instead of their uncompressed binary BVH, we use a four-wide BVH with quantized child node bounds, similar to Embree's QBVH8 layout [40]. We note that this choice of BVH was not motivated by any expected performance gain or memory use optimization, but rather because an easy-to-integrate library for this BVH was readily available. This BVH is built on the host using this library, after which it is uploaded to the GPU.

Though we use this reference method in our OptiX framework, the kernel itself does not use any OptiX constructs whatsoever, and could be used from arbitrary CUDA programs. To find the element containing the query point, the kernel performs a depth-first traversal using a software managed stack of BVH node references, immediately returning the element once it is found. Our implementation is similar to the pseudocode in Figure 3.

Extending our prior work [39], we modified our reference implementation to support data sets containing a mix of element types (i.e., those in Figure 2). Specifically, we have replaced the *pointInTetrahedra* test with a more general *pointInElement* test. Inside this function, we first determine the type of the given unstructured element based on the number of non-negative indices. From there, we call the respective intersection routine: Cramer's method for tetrahedra, and element-type-customized Newton-Raphson routines otherwise. By adding a conditional in this intersection test, we have observed a small but noticeable performance impact as a result. Although we could optimize this intersection routine to account for data sets containing only a limited subset of these unstructured element types, we chose not to pursue this optimization. Instead, we leave the overhead of this conditional to

```
rtDeclareVariable(Ray, ray, rtCurrentRay, );
rtDeclareVariable(float, prd, rtPayload, );
rtDeclareVariable(rtObject, world, , );
RT_PROGRAM void bounds(float *bounds, int elemID)
{ *bounds = box3f(vertex[index[elemID].x],...); }
RT_PROGRAM void intersect(int elemID) {
  if (intersectElement(ray.origin,elemID,result)
    && rtPotentialIntersection(1e-10f)) {
      prd = result; rtReportIntersection(0);
    }
}
__device__ float getSample(const vec3f P) {
  Ray ray(P, vec3f(1), 0, 0.f, 2e-10f);
  float prd_result = negInf;
  rtTrace(world, ray, prd_result,
     RT_VISIBILITY_ALL,
     RT_RAY_FLAG_TERMINATE_ON_FIRST_HIT
     |RT_RAY_FLAG_DISABLE_ANYHIT
     |RT_RAY_FLAG_DISABLE_CLOSESTHIT);
  return prd_result;
}
```

Fig. 4: *Pseudocode for our* `rtx-bvh` *method, which performs a point query by first tracing an infinitesimal ray from the point and then executing the point-in-element tests in the intersection program.*

isolate the performance of the hardware-acceleration from other factors across the data sets tested.

5 rtx-bvh: Exploiting RT Cores for BVH Traversal

The reference method is reasonably efficient, but it does not use the RT cores at all. To leverage these cores, we first have to reformulate our problem in such a way that it fits the hardware. In other words, we have to express point location as a *ray tracing* problem.

Staying conceptually close to our reference implementation, we can use OptiX to build an RTX BVH over the elements by creating an OptiX geometry with the given number of elements as custom, user-defined primitives. We can then write a bounding box kernel that computes each respective element's bounding box in parallel. To compute the bounds of a general unstructured element, we first consider the element type within the *bounding box program*, and read in that element's corresponding vertices. Since the faces of our elements are either linear or bilinear, we can compute an element's axis-aligned bounding box by iterating over all the vertices of the given element, computing the minimum and maximum corners. Once these boxes are computed and we have our geometry object, we can request OptiX to build an RTX acceleration structure over the elements of this geometry for us.

Although we are now armed with a hardware-accelerated BVH, one problem remains: the hardware knows only about tracing rays, not points. Thus, we must find a way to express our query points as "rays". Fortunately, we can view each query point as an infinitesimally short ray, and use an arbitrary direction (e.g., $(1,1,1)$) and a vanishingly small ray interval ($ray.tmax = 1e-10f$) to express this point to OptiX. When we trace such a "ray", the hardware will traverse the BVH and will visit every element potentially overlapping the ray to find an intersection. To find the element containing the query point, we attach an *intersection program* to our geometry that executes our point-in-element test and, when found, stores the intersected element ID in the per-ray data (Figure 4).

As the rays traced are vanishingly short, we can expect the traversal to visit roughly the same BVH nodes as our reference implementation, although with no guarantee that the hardware will visit *only* those nodes overlapping the point. Once the containing element is found we tell OptiX to report the hit, allowing the hardware to immediately terminate BVH traversal regardless of what else might be on the traversal stack, as done in the reference implementation. For performance reasons, we explicitly disable the *any-hit* and *closest-hit* programs to save the overhead of calling empty functions.

The `rtx-bvh` kernel defers the actual BVH construction and traversal to OptiX, which under the hood implements highly optimized BVH construction routines. During traversal, OptiX will automatically use hardware accelerated BVH traversal if available, and fall back to its own software traversal if not. Compared to the CUDA reference method, `rtx-bvh` leverages the ray tracing hardware to accelerate BVH traversal, although `rtx-bvh` still performs the point-in-element tests in software. Although traversing a ray is *more* expensive than traversing a point, the ray traversal is now hardware accelerated, and we can expect to observe a performance gain over the reference method.

6 Full Hardware Acceleration with RTX Triangles

Although the `rtx-bvh` method uses hardware-accelerated BVH traversal, it still relies on a software point-in-element test, limiting the potential speed-up it can achieve. To improve performance further, we must reduce these tests and eliminate the back and forth between the hardware traversal units and the programmable cores running the software point-in-element test. Our goal is to be able to make a single trace call and immediately get back just the ID of the element containing the point, with no software execution required in between.

To achieve this goal, we first note that each element is enclosed by a set of faces, meaning that any noninfinitesimal ray traced from a point within the element will hit one of these faces. Furthermore, if the faces of an element are planar, they can be accurately represented using triangles, and ray-triangle intersection is accelerated by RTX. In the planar case, we can represent an element by its tessellation, and instead of going back and forth between hardware BVH traversal and software intersection, we can let the hardware perform both the BVH traversal and ray-triangle intersection. When an intersection is found, we will be given the intersected triangle ID, which we can use to determine the corresponding element.

Before looking into the core problem addressed in this extension—namely, how to handle general elements with bilinear faces—let us first discuss the simplified case where all elements consist of only planar faces. In practice, many higher dimensional elements will twist and bend to better match the underlying data (meaning that some faces cannot be perfectly represented with a set of triangles) but for now, planar faces are easier to think about.

6.1 `RTX-Replicated-Faces`

Assuming all faces are planar, the most straightforward way to implement this ray-triangle-accelerated idea is to create a list of `int3` indices, one `int3` index for each triangular face defining an element, and two `int3` indices for each planar quadrilateral face defining an element. Then, in the *closest-hit* program we look up which element the hit triangle belongs to. In theory, that element *should* be the element that contains our point.

This technique is easy to implement, but in practice has some caveats. First, interior faces are now represented twice, and we need a way to ensure that the ray only reports the current element's face, and not the co-planar face from its neighbor. We solve this by constructing the triangles such that they always face inward towards the element, and trace the ray with back face culling enabled (Figures 5, 6a and 6b). Ray traversal, intersection, and back face culling are now all performed in hardware, and we can simply trace a ray and let the hardware do the work until the right face is found, eliminating the back and forth between hardware and software required by the previous methods.

Although the method as described so far works perfectly well for any query point inside a planar element, without further care it may return

```
struct Face { int3 index; int elemID; };
rtBuffer<Face, 1> faceBuffer;
rtDeclareVariable(float, prd, rtPayload, );
rtDeclareVariable(rtObject, world, , );
rtDeclareVariable(float, maxEdgeLength, , );
RT_PROGRAM void closest_hit() {
  const int faceID = rtGetPrimitiveIndex();
  const int elemID = faceBuffer[faceID].elemID;
  float fieldValue;
  if (interpolateElement(elemID, ray.origin, fieldValue)  )
    prd = fieldValue;
}
__device__ float getSample(const vec3f P) {
    Ray ray(P, vec3f(1), 0, 0.f, maxEdgeLength);
    float fieldValue = negInf;
    rtTrace(world, ray, fieldValue,
            RT_VISIBILITY_ALL,
            RT_RAY_FLAG_CULL_BACK_FACING_TRIANGLES
            |RT_RAY_FLAG_DISABLE_ANYHIT);
    return fieldValue;
}
```

Fig. 5: *Pseudocode kernels for our* `rtx-rep-faces` *method, which performs a point query by tracing a finite length ray and performing a single point-in-element test in the closest hit program.*

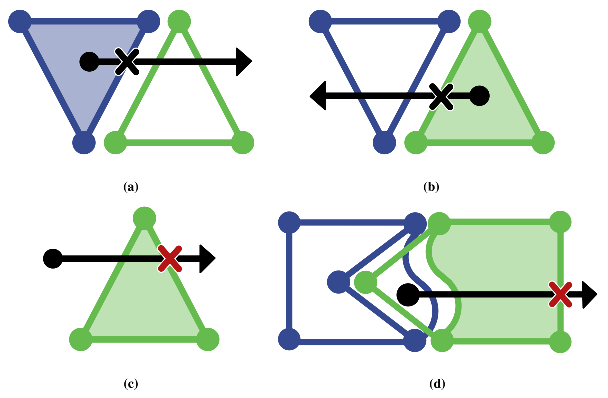


Fig. 6: *(a) and (b)* `rtx-rep-faces` *uses back face culling to avoid intersecting co-planar neighboring faces; (c) however, points outside any element can return false positive intersections as the exterior back faces are hidden, requiring an extra point-in-element test for correctness. In the case of bilinear faces, (d) triangle approximations result in false positive and negative intersections. Note that in (d), the S-like boundary is purely for illustration purposes, and in practice the face is bilinear with no inflection points. Also note that this is a 2D simplification of our* `rtx-rep-faces` *technique, and can be interpreted as a cross section of the 3D version.*

false positives for points outside the mesh. As shown in Figure 6c, a ray traced from a point outside the mesh can travel into an element and, with back face culling enabled, will not intersect the boundary face but rather the next interior face, incorrectly marking the point as contained in the boundary element. To ensure correctness in all planar face cases, we perform an additional point-in-element test inside the *closest-hit* program. Unlike the reference and `rtx-bvh` methods, this test needs to be done only once per ray, and thus is relatively cheap. When per-vertex scalar interpolation is desired, the barycentric coordinates computed during this point-in-element test are also needed for interpolation, and so we must compute this final point-in-element test anyway.

6.2 `RTX-Shared-Faces`

Instead of replicating shared faces as in the previous approach, a more memory efficient alternative is to find faces shared by neighboring

```
struct Face { int3 index; int2 elemIDs; };
rtBuffer<Face, 1> faceBuffer;
RT_PROGRAM void closest_hit() {
  int  faceID  = rtGetPrimitiveIndex();
  int2 elemIDs = faceBuffer[faceID].elemIDs;
  int  elemID  = rtIsTriangleHitBackFace() ?
                 elemIDs[1] : elemIDs[0];
  if (elemID < 0) return;
  // store ID or compute scalar field ...
}
```

Fig. 7: *The* closest_hit *program for* `rtx-shrd-faces`. (getSample() *is the same as in Figure 5).*

elements and merge them in a preprocess. Although this preprocessing step can be expensive, the benefits are significant: the resulting output triangle mesh is much smaller, and no longer requires special treatment to cull co-planar duplicate faces.

For each face, we now store two integers, which specify the IDs of the elements on its front and back face (or -1 if no element exists on that side). In the *closest-hit* program, we check to see if our ray hit a front face or a back face, and use that information to determine the containing element ID (see Figures 9a and 9b). As back face culling is no longer needed to hide co-planar faces, `rtx-shrd-faces` eliminates the caveats of `rtx-rep-faces` discussed above (e.g., for points outside the volume, Figure 9c). Pseudocode for `rtx-shrd-faces` is shown in Figure 7.

During our preprocessing step, we compute a list of unique faces, where we tag the front and back sides of each face with a corresponding element ID (Figure 8). To match these faces together, we first define a *unique representation* of a face by temporarily sorting its vertex indices and hashing them (see `hash_tri` in Figure 8). We can then use this hash to find faces shared by elements using a hash map, although we preserve the original vertex order during insertion for later rendering purposes.

First, we loop through the list of inside faces of each element computed in Section 6.1 and find the hash of each face's unique representation. We then use this hash to check in a hash map if we have already added this face to our list of shared faces. If the face is not in the map, we first add the face to our shared faces list. At this point in our preprocessing, we know which element the current replicated face belongs to and can also conclude the face is oriented toward the current element. Therefore, we set the front face ID of the newly added face to be the current element ID and initialize the back face ID to -1. Finally, we insert the face into the hash map along with the face's index in the shared faces list.

If the face is already in the map it must have been inserted by another element. In this case, the previously inserted face is shared with the current element and is oriented away toward the other element. We use the hash map to determine where the shared face is located in our shared faces list, and set the current element ID as the back face ID for the shared face.

7 Extending to General Elements with Bilinear Faces

As described so far, our triangle-based methods will work well for data sets containing elements with planar faces. The element location process is performed entirely in hardware, allowing us to provide a point and get back the containing element without any software intervention. However, not all general unstructured elements have planar faces. For elements with quad faces (i.e., pyramids, wedges, hexahedra), if the four vertices forming a quad face do not lie on the same plane, we can no longer accurately represent the face using two triangles. Such

```
struct Face { int3 index; int2 elemIDs; };
std::vector<Face> shared_tris;
size_t hash_tri(int3 tri) { return hash(sort(tri)); };
void compute_shared_tri_faces(UnstructuredMesh &mesh) {
  std::unordered_map<int3, int, hash_tri> tri_id_map;
  for (auto &element in mesh.elements)
    for (auto &triangle in element.inside_facing_triangles)
      if (!tri_id_map.find(triangle)) {
        // This element inserts the face, we are front side
        int triangle_index = shared_tris.size();
        tri_id_map[triangle] = triangle_index;
        Face new_face = Face(triangle, int2(element.ID, -1));
        shared_tris.push_back(new_face)
      } else {
        // Face was already inserted, we are on the back side
        Face &tri = shared_tris[tri_id_map[triangle]];
        tri.elemIDs[1] = element.ID;
      }
}
```

Fig. 8: *Pseudocode to compute a list of shared faces, where each face stores the IDs of the elements on the front and back side.*

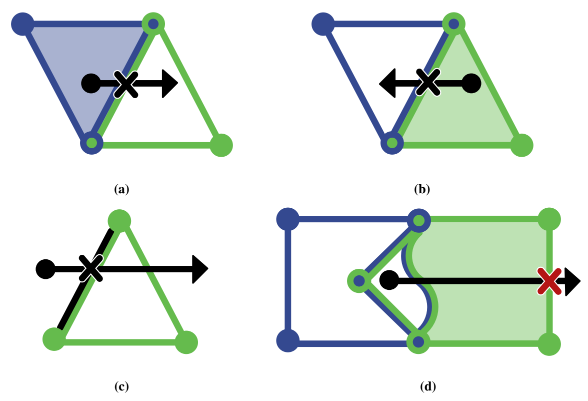


Fig. 9: *(a,b) unlike* `rtx-rep-faces`*, the* `rtx-shrd-faces` *approach does not rely on back face culling and replicated faces, and instead relies on face orientation to determine the containing element. (c) Since we no longer rely on backface culling, we now get correct results for points outside an element. (d) Still, we run into issues near bilinear faces with approximate tessellations. (Note again that in (d), the S-like boundary is purely for illustration purposes, and in practice is bilinear with no inflection points.)*

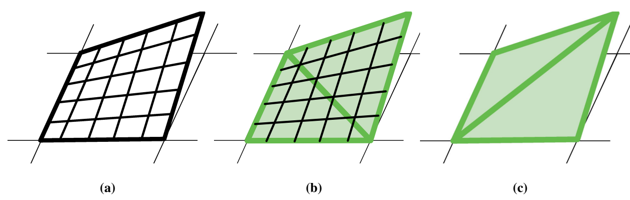


Fig. 10: *General unstructured elements can contain curved bilinear faces such as (a). Although the edges of these patches are linear, the interpolated surface bends in the center to meet all four possibly non-coplanar corners. These bilinear faces can be approximately tessellated in two different configurations. Tessellation (b) lies entirely below the surface, whereas tessellation (c) lies entirely above. We refer to (b) as an underestimating tessellation and (c) as an overestimating tessellation.*

non-planar faces are represented as curved bilinear surfaces, which we extend our method to support in this section. To do so, we build on the following observations of the problems that arise when applying our triangle-based methods to a data set containing curved bilinear faces.

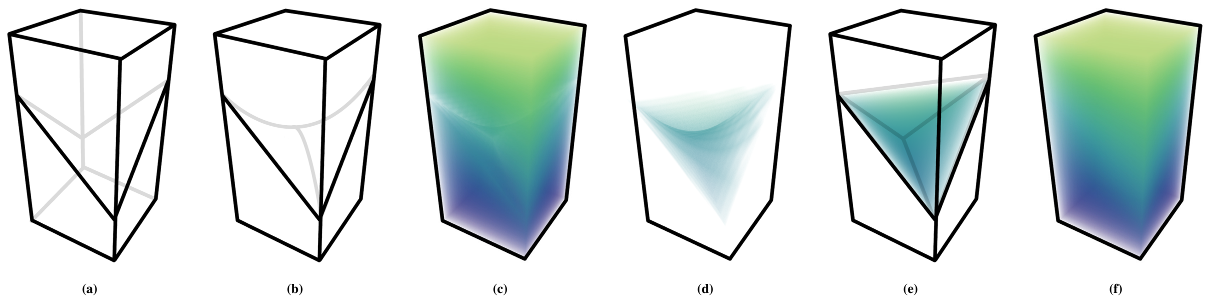


Fig. 11: *When rendering general bilinear cells using our* `rtx-shrd-faces` *approach, artifacts can occur at bilinear faces. In (a), we place two hexahedra together, such that a bilinear face (b) separates the two cells vertically. As shown in (c), artifacts occur near this face due to false positive and negative point query intersections. However, these artifacts are isolated to the bounding tetrahedra for that bilinear patch, as shown in (d). With* `rtx-shrd-bilinear-faces`*, these faces are accounted for, correcting these artifacts (e,f).*

1. **Curved bilinear faces cannot be accurately triangulated**. The triangle based methods described so far rely on the assumption that we can accurately represent the faces of an unstructured element using only triangles. However, this no longer holds for curved bilinear faces. Turing's RT cores do not support bilinear surface intersections, and moving the tests back to software would lose the hardware acceleration benefits gained in the previous sections. Although the surfaces could be approximated by more than two triangles, this would impact performance, increase memory usage, and it is unclear how many triangles would be needed to accurately represent each face.

2. **Bilinear surfaces satisfy the "Strong Convex Hull" property**. This property guarantees that each bilinear surface is entirely contained within the convex hull defined by the control points of that surface. In the case of bilinear faces, the convex hull is the combination of the under and overestimating tessellations of the bilinear surface's quad, shown in Figure 10.

3. **We can isolate problematic cases by generating convex hulls containing our bilinear faces**. Depending on the direction of the ray, our methods described so far can result in either false positive or negative intersections (Figure 9d) when the query point falls inside a bilinear surface's convex hull. These false positive and negative intersections result in visual artifacts as shown in Figure 11. However, for query points outside the bilinear face's convex hull, the `rtx-rep-faces` and `rtx-shrd-faces` methods locate the correct containing element.

4. **The convex hull of a bilinear surface is a tetrahedron**, and as we have demonstrated with both `rtx-rep-faces` and `rtx-shrd-faces` methods, we can accelerate tetrahedral element point location using the RT cores. Thus, we can form the tetrahedra bounding each bilinear face and, when a point query falls inside one, perform a correct bilinear surface intersection test to determine the side the point falls on to find the correct containing element.

7.1 `RTX-Shared-Bilinear-Faces`

Using the above observations, we can extend our `rtx-shrd-faces` approach to support unstructured elements containing bilinear faces. First, as in the shared faces approach, we store two integers per bilinear face. These integers specify the IDs of the elements on the front and back sides of the bilinear face (or -1 if no element exists on that side). However, unlike the triangle faces, we do not insert these shared bilinear faces into our RTX acceleration structure directly.

Instead, for each shared bilinear face, we generate its bounding tetrahedron to isolate problematic cases. When we generate the triangle faces to represent this bounding tetrahedron we also store two integers per triangle, that are simply copied from the shared bilinear face. For each triangle, we additionally store the fourth vertex index required to reconstruct the original shared bilinear surface. During traversal we can now perform hardware accelerated intersection tests against the faces of the bounding tetrahedra of each bilinear face, rather than falling back to a software intersection program to test against the bilinear surfaces.

Next, we need to be able to detect if a query point lies within a bilinear face's bounding tetrahedron, or just a regular tetrahedral element. As our `rtx-shrd-faces` approach no longer culls back faces, we can leverage the face orientation (in addition to the presence of a fourth vertex) to detect if a point is located within a bilinear face's bounding tetrahedron. This requires us to reorient the triangles of the bounding tetrahedra to face outwards away from the bilinear surface, as opposed to the inwards facing triangles of regular tetrahedral elements. Shared element IDs are then adjusted accordingly so that the correct element ID is returned depending on the side of the face our ray hits. (see Figure 14). Following from observation 3, we can then use the fact that our point lies within a bilinear face's bounding tetrahedron to determine that we need to do additional intersection testing in software to properly handle the bilinear face.

In the *closest-hit* program, if OptiX reports we hit either a front face or a back face not belonging to a bounding tetrahedron, we can safely conclude the element that our ray origin lies within, without considering any bilinear faces (see Figure 14a). On the other hand, if we hit the back face of a triangle that belongs to a bounding tetrahedron, we know that the ray origin must lie within that bounding tetrahedron, and that further intersection testing is required to handle the corresponding bilinear face. In this case, we use the fourth vertex to reconstruct that bounding tetrahedron's corresponding bilinear surface and test to see if we hit that surface in the *closest-hit* program, as shown in Figure 14b. To perform this bilinear surface intersection test, we use the GARP intersector from Reshetov [29]. If the ray intersects the surface (Figure 14c), we can conclude the point is in the back facing element of the closest hit triangle, or outside the volume if there is no neighbor. If the ray does not hit the surface (Figure 14d), the query point must be inside the front facing element of the closest hit triangle.

With this approach, we need at most one bilinear surface intersection test per point query, and more often than not we can skip performing bilinear surface intersections entirely. For bilinear surfaces that are nearly planar, the corresponding bounding tetrahedra will decrease in volume, and thus our slightly more expensive final intersection routine will be less likely to be called.

We integrate the generation of the triangulated bounding tetrahedra within the shared faces preprocessing step. We define the unique

```
struct Face { int4 index; int2 elemIDs; };
rtBuffer<Face, 1> faceBuffer;
RT_PROGRAM void closest_hit() {
  int  faceID = rtGetPrimitiveIndex();
  int2 elemIDs = faceBuffer[faceID].elementIDs;
  bool isBilinear = faceBuffer[faceID].w != -1;
  bool hitBackFace = rtIsTriangleHitBackFace();
  int  elemID  = hitBackFace ? elemIDs[1] : elemIDs[0];
  if (hitBackFace && isBilinear)
    if (intersectBilinearPatch(faceID, ray))
      elemID = elemIDs[0];
  if (elemID < 0) return;
  // store ID or compute scalar field ...
}
```

Fig. 12: *The* closest_hit *program for* `rtx-shrd-faces` *, now modified to support elements with bilinear faces.* getSample() *is the same as in Figure 5. Note that the previous* `int3` *index is now an* `int4`*, potentially representing a counter-clockwise quad represented by the* `int3` *triangle face we hit.*

```
// Below similar to Fig 7
void compute_shared_faces(UnstructuredMesh &mesh) {
  compute_shared_tri_faces(mesh);
  compute_shared_quad_faces(mesh);
  // Tessellate quads, adding to shared tris list
  for quad in shared_quads:
    if (quad.is_planar) {
      Face faces[2] = tesselate_planar(quad)
      shared_tris.push_back(faces)
    else {
      // both over and underestimating tris needed
      // for bilinear patches. Overestimating tris have
      // swapped elemIDs.
      Face faces[4] = tesselate_bilinear(quad)
      shared_tris.push_back(faces)
    }
}
```

Fig. 13: *Preprocessing code for bilinear shared faces. Shared planar quads generate two shared triangle faces whereas shared bilinear quads generate four.*

representation of each bilinear face by sorting its four vertex indices and hashing them as before. The list of shared bilinear faces with front and back element IDs are generated as before, using the unique representation to check which element is the first and second to insert the face into the list.

After computing the list of shared bilinear faces, we compute a set of triangles used to represent the bounding tetrahedron for each bilinear face. In the case of planar bilinear faces, we instead generate two planar triangles as done in the planar `rtx-shrd-faces` approach. (Figure 13). When the bounding tetrahedron triangles are generated they are made to face away from the bilinear face that the bounding tetrahedron contains. These bounding tetrahedron's triangles are tagged as belonging to a bilinear face. This orientation and tag together allows us to detect within the *closest-hit* program whether a point query lies within a bounding tetrahedron or not. (See Figure 12) The triangles are then inserted into the list of shared triangles, placing them into the same BVH as those representing faces of elements with planar faces.

8 Common Implementation Details

Once the set of triangles is generated for each method, the actual OptiX set-up code is almost identical. We create an OptiX triangle geometry for the triangle mesh and assign the triangle vertices and indices. For each triangle, we use an `int4` to store the indices, where the first three indices reference a counterclockwise triangle. If the triangle comes from a bounding tetrahedron of a bilinear surface, the fourth index is

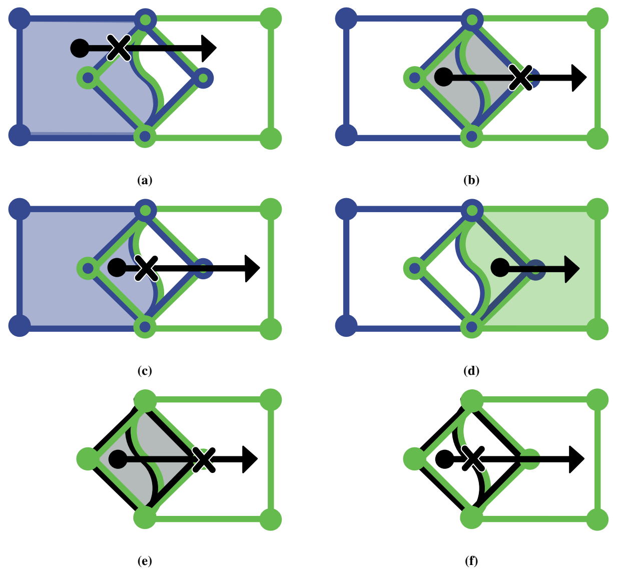


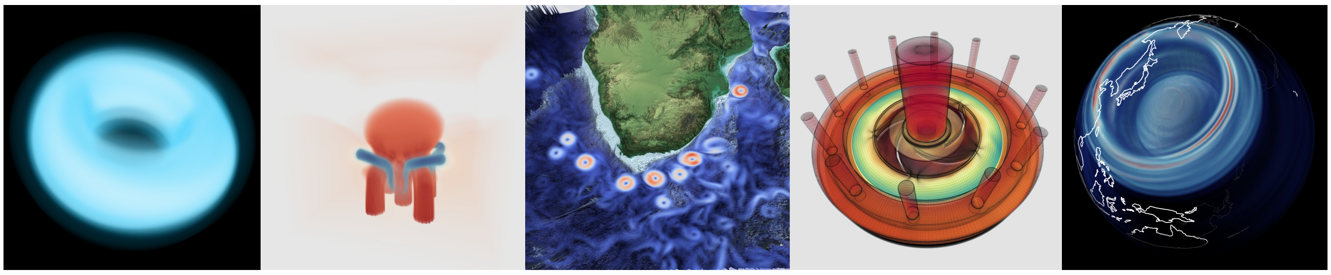
Fig. 14: *(a) As in the original* `rtx-shrd-faces` *approach, when a front face is hit we return the corresponding front element ID. However, back face hits belonging to bilinear patches (b) require additional information to conclude which element contains the point. If we hit the patch associated with the current face (c), the point is in the back face element. Otherwise, (d) the point is in the front face element. This same test can be used to resolve point queries outside an element (e,f) near bilinear boundaries. (Note that the S-like boundary is purely for illustration purposes, and in practice is bilinear with no inflection points.)*

set to the missing quad vertex, such that the set of vertices in order 1, 2, 3, 4 forms a counterclockwise loop around the quad. Otherwise, this fourth index is set to -1, indicating the triangle does not belong to a curved bilinear surface. This triangle geometry is then placed in an optix acceleration structure. After that, we upload the buffer of unstructured elements, as well as a buffer containing with either one or two element IDs per-triangle. As an optimization, for all methods we explicitly disable the *any-hit* program. This guarantees to the ray tracer that it can skip the *any-hit* program, avoiding any back and forth between hardware traversal and an empty software *any-hit* program.

A key difference of our triangle-based methods when compared to `rtx-bvh` is that we can no longer use an infinitesmal ray length, since such short rays would not reach the faces. Although infinite length rays would intersect the faces, they would necessarily require the hardware to perform more traversal operations, which, even with hardware acceleration, is expensive. To address this problem, we compute the maximum edge length of any element in the data set, and use this length as the ray's `ray.tmax` value. This approach ensures that rays can reach the right faces, while limiting the traversal distance. Finally, although an arbitrary ray direction of, e.g., $(1,1,1)$ works for most point queries, we have encountered rare artifacts when rays glance triangle edges. To mitigate these artifacts, we choose a random direction for each ray in our volumetric ray caster example.

9 Evaluation

Given these four kernels, we can now evaluate their relative performance. We ran our experiments on a mid-range workstation with an Intel Core i7–5960X CPU and 128 GBs of RAM and tested on a variety of both consumer and high-end RTX-enabled cards. Specifically, we use an RTX 2080, an RTX 2080TI, a Titan RTX, and an RTX 8000. For reference, we also ran our experiments on a pre-Turing Titan V,



Fusion
3 M tets, 622 K verts
scalar per-vertex

Jets
12 M tets, 2.1 M verts
scalar per-vertex

Agulhas Current
35.7 M tets and 6.2 M verts
14.4M wedges,
scalar per-cell, + 10.1 M tris

Scivis 2011 Pump
6.4 M hexes and 6.6 M verts,
scalar per-vertex, + 963 K tris

Japan Earthquake
62.2 M tets and 15 M verts
8M hexes,
scalar per-vertex, + 257 K tris

Fig. 15: *Sample images and statistical data for the data sets used in evaluating our kernels.* Agulhas*,* Scivis 2011 *and* Japan *include triangle meshes for the bathymetry, wireframe, and continent outlines, respectively. These are used during rendering but do not affect the point query kernels.*

which performs all ray tracing operations in software.

We ran all our evaluations on Ubuntu 18.04 using Optix 6.5, with Nvidia driver version 440.44 and CUDA 10.2. The data sets used for tetrahedral point query evaluation cover a range of shapes and sizes (see Figure 15), from 3 to nearly 63 million elements. All but the *Jets* data set are sparse, in that only part of the data's bounding box is covered by unstructured elements. For *Fusion*, cells cover only the torus; in *Agulhas*, cells cover only "wet" cells (roughly 50% of the bounding box; and for *Japan*, our data includes only nonzero elements, covering just 5.15% of the bounding box.

To test our newly supported general unstructured elements, we additionally chose to use two variants of the *Agulhas* and *Japan* data sets. The original *Agulhas* data set is composed solely of wedges, and the *Japan* data set is composed of hexahedra. By additionally using the original forms of these data sets, we can compare the performance of our approach for general unstructured meshes directly against our previous tet-mesh-only approach. Finally, as an additional point of comparison against other unstructured volume rendering solutions, we include the *Scivis 2011 Pump* data set, which is composed of 6.4 M hexahedra.

9.1 Memory Usage

We first measured the total GPU memory usage for the various methods, listed in Table 1. We observe that on Turing, our kernels require significantly less GPU memory than on Volta, especially for the triangle-based variants. On our `rtx-bvh` method, we see a 31% overall decrease in peak memory usage when moving from Volta to Turing. Likewise, our peak memory usage on our `rtx-shrd-faces` approach sees a 59% overall decrease in memory usage.

Irrespective of the GPU architecture, we found that OptiX 6.5 exhibited a significant difference between its *final* memory usage after it had finished building all data structures, compared to its *peak* memory usage while building these data structures. Although this overhead is temporary, it was significant enough that some of our experiments initially ran out of memory on the RTX 2080. To avoid OptiX allocating such a large block of scratch memory all at once, we split the mesh elements ahead of time into smaller groups, and serialized BVH construction over these groups.

We first partition the set of primitives into groups of at most 1 million each, and then put each group into a geometry instance with a corresponding acceleration structure. These geometry instances are then put into a "top-level" OptiX acceleration structure. Since each BVH is now much smaller, the peak memory usage during construction is lower, allowing even the 8 GB card to fit all but one experiment. Turing architecture GPUs support this two-level data structure natively in hardware, so this two-level approach does not significantly impact performance (some experiments even performed marginally better). Additionally, it is worth noting that OptiX 7 allows for much more

	Volta, no RTX						**Turing, with RTX**					
model	fusion	jets	agul-t	jpn-t	agul-w	jpn-h	fusion	jets	agul-t	jpn-t	agul-w	jpn-h
#elements	3M	12M	36M	62M	14M	8M	3M	12M	36M	62M	14M	8M
`cuda-bvh` (Section 4)												
final	725M	921M	2.0G	3.2G	1.3G	1.0G	466M	844M	1.9G	3.1G	1.3G	1.0G
`rtx-bvh` (Section 5)												
peak (no p/s)	837M	2.4G	6.3G	10.6G	3.0G	2.0G	656M	2.1G	5.7G	9.6G	2.6G	1.6G
peak (w/ p/s)	725M	1.6G	3.9G	6.5G	2.5G	1.7G	504M	1.1G	2.1G	4.4G	1.8G	1.4G
final	717M	1.6G	3.8G	6.1G	2.5G	1.7G	464M	754M	1.7G	3.1G	1.8G	1.3G
`rtx-rep-faces` (Section 6.1)												
#faces	11.9M	49.1M	143M	249M	NA	NA	11.9M	49.1M	143M	249M	NA	NA
peak (no p/s)	2.5G	9.0G	`(oom)`	`(oom)`	NA	NA	1.6G	5.9G	16.9G	`(oom)`	NA	NA
peak (w/ p/s)	2.1G	7.3G	`(oom)`	`(oom)`	NA	NA	1.2G	2.3G	6.1G	11.0G	NA	NA
final	2.1G	7.2G	`(oom)`	`(oom)`	NA	NA	770M	1.8G	5.4G	10.7G	NA	NA
`rtx-shrd-faces` (Section 6.2)												
#faces	5.99M	24.7M	72M	134M	58M	52M	5.99M	24.7M	72M	134M	58M	52M
peak (no p/s)	1.5G	4.9G	`(oom)`	`(oom)`	11G	9.5G	960M	3.3G	9.3G	16.9G	9.0G	8.2G
peak (w/ p/s)	1.3G	4.1G	11.3G	`(oom)`	8.6G	7.8G	846M	1.7G	4.4G	7.2G	4.0G	3.7G
final	1.3G	4.0G	11.3G	`(oom)`	8.6G	7.8G	643M	1.4G	3.9G	6.8G	3.8G	3.3G

Table 1: *GPU memory cost for our four kernels across our test data sets. "Peak" is the peak memory used by OptiX during the BVH build (with and without pre-splitting); "final" is the total memory required after BVH construction. Additional nonvolume data, e.g., framebuffer and surface meshes, are not included. (-t indicates a data set contains tetrahedra, -w contains wedges and -h contains hexahedra)*

	Synthetic Uniform (samples/sec)					
	fusion	jets	agulh	jpn-tet	agul-wed	jpn-hex
#elements	(3M)	(12M)	(36M)	(62M)	(14M)	(8M)
Titan V						
`cuda-bvh`	89.7M	1.55G	971M	461M	358.7M	183.0M
`rtx-bvh`	91.8M	1.05G	741M	373M	179.6M	200.5M
`rtx-rep-faces`	34.7M	407M	`(oom)`	`(oom)`	NA	NA
`rtx-shrd-faces`	59.7M	689M	397M	`(oom)`	238.6M	150.1M
RTX 2080						
`cuda-bvh`	53M	996M	563M	263M	121.0M	136.1M
`rtx-bvh`	98.2M	1.17G	1.03G	525M	115.0M	175.0M
`rtx-rep-faces`	253M	1.23G	1.11G	`(oom)`	NA	NA
`rtx-shrd-faces`	354M	1.62G	1.58G	1.28G	1.3G	960.2M
RTX 2080 TI						
`cuda-bvh`	75.8M	1.3G	764.6M	360.5M	180.1M	190.8M
`rtx-bvh`	129.2M	1.5G	1.3G	672.6M	170.4M	235.9M
`rtx-rep-faces`	354.4M	1.8G	1.6G	1.3G	NA	NA
`rtx-shrd-faces`	492.8M	2.3G	2.3G	1.8G	1.8G	1.3G
Titan RTX						
`cuda-bvh`	84.5M	1.39G	813M	386M	188.7M	202.5M
`rtx-bvh`	145M	1.67G	1.43G	736M	178.3M	259.5M
`rtx-rep-faces`	377M	1.78G	1.7G	1.4G	NA	NA
`rtx-shrd-faces`	537M	2.4G	2.4G	2.0G	1.9G	1.3G
RTX 8000						
`cuda-bvh`	83.2M	1.3G	768M	371M	182M	195.7
`rtx-bvh`	134.7M	1.6G	1.3G	699.4M	173M	248.8M
`rtx-rep-faces`	363M	1.8G	1.6G	1.4G	NA	NA
`rtx-shrd-faces`	505M	2.3G	2.3G	1.9G	1.9G	1.3G

Table 2: *Synthetic uniform performance results for all our kernels, across all data sets. Here we execute point queries in a coherent fashion, one point query per element. All experiments use pre-splitting (see Section 9.1), and are averaged across several runs to reduce launch overhead.* `(oom)` *indicates OptiX ran out of memory during the BVH build.*

	Synthetic Random (samples/sec)					
	fusion	jets	agul	jpn-t	agul-w	jpn-h
#elements	(3M)	(12M)	(36M)	(62M)	(14M)	(8M)
Titan V						
`cuda-bvh`	36.4M	82.4M	83.8M	70.3M	130.1M	73.4M
`rtx-bvh`	30.2M	108M	83.6M	68.6M	80.3M	69.8M
`rtx-rep-faces`	23.7M	81.5M	`(oom)`	`(oom)`	NA	NA
`rtx-shrd-faces`	35.1M	101M	63.6M	`(oom)`	78.5M	51.1M
RTX 2080						
`cuda-bvh`	19.7M	60.5M	53.3M	44.1M	52.8M	50.5M
`rtx-bvh`	24.7M	74.7M	69M	59.6M	61.1M	52.6M
`rtx-rep-faces`	65.2M	159M	126M	`(oom)`	NA	NA
`rtx-shrd-faces`	76.1M	175M	130M	100M	125.3M	99.3M
RTX 2080 TI						
`cuda-bvh`	27.6M	82.2M	69.9M	71.3M	81.7M	71.4M
`rtx-bvh`	30.6M	99.1M	82.7M	89.6M	90.7M	74.2M
`rtx-rep-faces`	92.5M	227.9M	166.8M	129.0M	NA	NA
`rtx-shrd-faces`	109.4M	245.6M	166.6M	141.0M	184.5M	142.6M
Titan RTX						
`cuda-bvh`	31.4M	90.1M	77.5M	62.8M	85.2M	75.5M
`rtx-bvh`	37.1M	111M	99.5M	83.9M	96.2M	80.2M
`rtx-rep-faces`	99M	243M	188M	138M	NA	NA
`rtx-shrd-faces`	116M	268M	196M	150M	195.8M	150.9M
RTX 8000						
`cuda-bvh`	31.1M	89.9M	76.4M	62.4M	82.9M	72.3M
`rtx-bvh`	35.7M	109.2M	98.5M	83.1M	93.2M	79.7M
`rtx-rep-faces`	99.3M	244.4M	188.2M	137.5M	NA	NA
`rtx-shrd-faces`	116.6M	269M	196.5M	150.2M	195.4M	149.5M

Table 3: *Synthetic random performance results for all our kernels, across all data sets. Here we execute 10M point queries per launch, each query originating in a randomly chosen cell at a random location inside that cell. All experiments use pre-splitting (see Section 9.1), and are averaged across several runs to reduce launch overhead.* `(oom)` *indicates OptiX ran out of memory during the BVH build.*

fine-grained control over this BVH construction process than OptiX 6.5. There are likely more ways to minimize this peak memory usage than what we have explored so far.

Since we are now able to support general unstructured elements, we can save a large amount of memory for data sets that were previously being tetrahedralized. In practice, many unstructured volumes use these general unstructured elements as the basis for more efficient and numerically stable simulations. However, there are potential memory benefits to using higher dimensional elements as well, since fewer elements are required to represent the same data set. For example, two of our data sets, *Agulhas* and *Japan*, were previously general unstructured meshes before we tetrahedralized them in our prior work [39]. On the other hand, if a general unstructured mesh is tetrahedralized, the number of elements increases significantly. Pyramids require two tetrahedra, wedges require three, and hexahedra commonly use six.

In our example use case, we find this can make the difference between a mesh fitting in memory or not. For example, on our Volta experiments, we were unable to fit the tetrahedralized *Japan* data set (62.2 M tetrahedra) into memory with our `rtx-shrd-faces` approach. However, the hexahedral *Japan* data set (8 M hexahedra) fits comfortably within the same 12 GB card. With the *Japan* data set, we see, on average, a 61% overall reduction in peak memory usage when comparing the tetrahedralized version to the original hexahedral one. Likewise, we find an average 20% reduction in memory usage on the *Agulhas* data set when comparing the tetrahedralized version (35 M tetrahedra) to the original one with wedges (14 M wedges). Note that for these comparisons, our wedges and hexes used eight indices per element, whereas tetrahedra used only four indices. For the wedges case, we would likely see larger memory improvements by using six indices instead of eight.

9.2 Benchmark Performance

To measure just the raw query performance of our kernels, we conducted a set of synthetic benchmarks (Tables 2 and 3). We first performed these benchmarks by taking uniformly and randomly distributed sample points within the volume's bounding box; however, as most models are sparse, many of these samples will not be inside any element, making them artificially less expensive to compute. This distribution led to an unrealistically high average sampling rate for the kernels.

Such a purely spatial sample pattern is not entirely unrealistic; in fact, our prototype volume renderer will generate these kinds of distributions in the next section. Nevertheless, we felt this bounding-box-based sampling pattern was artificially inflating our sampling rates since rays that fall outside the volume quickly terminate during BVH traversal. We decided to change this sampling pattern to instead always place samples within valid elements. The *uniform* benchmark launches one thread per cell in a coalesced order and takes a sample at the element's center. The *random* benchmark has each thread select a random cell, and sample an arbitrary position within the cell. For general unstructured cells, some of these randomized positions may fall outside elements near bilinear boundaries. However, this is not necessarily an issue, since this approach covers both inside and outside bilinear intersection cases described in Figure 12.

On the pre-Turing Titan V, we see that, as expected, performance decreases as we increasingly use more ray tracing. Tracing a ray is inherently more expensive than querying a point, and without hardware acceleration, our point queries will be slower. Despite this theoretically higher cost, when we ran our benchmarks on GPUs with hardware-accelerated ray tracing, our kernels not only performed well compared to the reference method, but outperformed it significantly–by $1.6-6.6\times$ for tetrahedral meshes on the *uniform* benchmark and $2-4\times$ on the *random* benchmark. Our general unstructured meshes also see significant performance improvements–they are $6.4-11\times$ faster than the reference method on the *uniform* benchmark and $2-4\times$ faster on the *random* benchmark.

Since Newton-Raphson-based bilinear point queries traditionally perform worse than Cramer's rule-based tetrahedral point queries, we also wanted to compare the differences in performance between our general unstructured meshes and their tetrahedralized versions. As we anticipated, our baseline method on the *uniform* benchmark exhibits a decrease in performance when comparing our tetrahedralized meshes against their original hexahedral and wedge-based versions–by about $3.21-3.65\times$ on the *Agulhas* data set and by $0.86-1.52\times$ on the *Japan* benchmark. However, after moving to our hardware-accelerated ray tracing kernels, this penalty is reduced dramatically to only $0.21-0.27\times$ for *Agulhas* and $0.33-0.46\times$ for *Japan*. On our *random* benchmark, results are less clear, and we see certain cases where performance decreases slightly, with other experiments we see a slight increase in performance. However, for all these *random* benchmarks, we see little to no difference in performance between the tetrahedralized meshes and their original hexahedral- and wedge-based versions.

An interesting outlier in these results is our smallest data set, *Fusion*, which sees the worst absolute performance on the synthetic benchmarks. The tetrahedra in the *Fusion* data set have a much wider difference between the minimum and maximum edge lengths, and are densely packed around the center line of the torus. As the ray-tracing-based methods use the maximum edge length as the ray query distance, rays will traverse many more BVH nodes than required on the *Fusion* data set compared to the other data sets, impacting performance on the synthetic benchmarks.

On the *random* benchmark, we find that, as anticipated, the lack of query point coherence between neighboring threads impacts performance on all the kernels evaluated. Across all the methods, we see a decrease in performance on the order of $5-10\times$; however, our ray-tracing-accelerated point queries continue to outperform the reference. The *uniform* benchmarks achieve much higher sample rates on all methods, with our `rtx-shrd-faces` kernel reaching on the order of 1–2 billion samples per second. This result far exceeded our expectations, as the rays are by no means coherent from a rendering sense. Our experimental setup guarantees that even in the *uniform* case, no two rays will ever hit the same face.

9.3 Unstructured Volume Ray Marching

To see how these speed-ups translate to a more challenging application, we implemented a prototype volume ray marcher similar to that presented by Rathke et al. [28]. For each pixel, we march a ray through the volume's bounding box and perform point queries at a fixed step size, sampling the volume approximately once per element. At each sample point, the renderer uses one of our point-query kernels to compute the scalar field value. In contrast to unstructured volume renderers based on marching from element to element (e.g., [18, 19]), the renderer uses a fixed sampling rate along the ray and is not guaranteed to sample each element, though can achieve better performance as a result. The field value is then assigned a color and opacity from a transfer function stored in a 1D texture. The color-mapped samples are accumulated along the ray until the ray's opacity exceeds 99%. If a sample falls outside the volume, the ray marcher treats the sample as being fully transparent. If the data set includes surface geometry, we place the geometry into an OptiX triangles geometry instance and first trace a ray to find the nearest surface intersection point before integrating the volume up to the surface.

We find that the speed-ups achieved by our kernels on the synthetic benchmarks carry over to rendering. Our fastest method, `rtx-shrd-faces`, achieves a $1.5-4\times$ speed-up over the reference on tetrahedral

	Volume Rendering (FPS, 1024^2 pix)					
	fusion	jets	agulh	jpn-tet	agul-wed	jpn-hex
#elements	(3M)	(12M)	(36M)	(62M)	(14M)	(8M)
Titan V						
`cuda-bvh`	13.98	27.64	24.62	5.15	6.68	1.66
`rtx-bvh`	5.74	13.7	17.3	3.07	7.35	2.13
`rtx-rep-faces`	5.82	8.79	`(oom)`	`(oom)`	NA	NA
`rtx-shrd-faces`	9.4	13.2	`(oom)`	`(oom)`	11.2	2.22
RTX 2080						
`cuda-bvh`	8.85	17.2	19.6	3.18	2.49	1.34
`rtx-bvh`	6.45	9.78	13.1	3	4.55	2.28
`rtx-rep-faces`	21.6	22.5	28.3	`(oom)`	NA	NA
`rtx-shrd-faces`	33.7	29.7	35.4	5.53	37.0	8.89
RTX 2080 TI						
`cuda-bvh`	10.78	22.4	24.3	4.31	3.53	1.85
`rtx-bvh`	4.07	7.01	13.4	2.32	6.30	2.84
`rtx-rep-faces`	28.42	30.0	36.2	7.72	NA	NA
`rtx-shrd-faces`	41.78	36.2	46.1	9.57	46.5	11.2
Titan RTX						
`cuda-bvh`	12.7	22.6	26.0	4.59	3.87	2.05
`rtx-bvh`	4.97	8.39	16.2	2.55	6.66	3.36
`rtx-rep-faces`	27.7	31.7	37.8	8.20	NA	NA
`rtx-shrd-faces`	42.0	38.6	48.8	10.1	48.4	12.1
RTX 8000						
`cuda-bvh`	12.3	22.5	25.5	4.41	3.67	1.91
`rtx-bvh`	5.05	8.33	15.8	2.46	6.52	3.32
`rtx-rep-faces`	27.9	31.5	37.8	8.13	NA	NA
`rtx-shrd-faces`	41.9	39.0	48.8	10.0	47.3	11.5

Table 4: *Volume rendering performance results for all our kernels, across all data sets. A series of view aligned point queries are executed within a volumetric ray caster, producing the images seen in Figure 15. All experiments use pre-splitting (see Section 9.1), and are averaged across several runs to reduce launch overhead.* `(oom)` *indicates OptiX ran out of memory during the BVH build.*

	Volume Rendering (FPS, 1024^2 pix)						
	fusion	jets	agul	jpn-t	agul-w	jpn-h	scivis-h
#elements	(3M)	(12M)	(36M)	(62M)	(14M)	(8M)	(6M)
Paraview	0.80	2.1	0.04	0.03	0.04	0.04	0.05
OSPRay	1.64	0.76	0.58	0.48	0.93	1.94	0.23
IndeX	40.9	6.7	NA	8.10	NA	7.83	13.0
Ours	41.9	39.0	48.8	10.0	47.3	11.5	25.6

Table 5: *A set of rendering performance comparisons against other common unstructured volume rendering frameworks. OSPRay volume rendering performance was measured on an i9-9920X. All other measurements were recorded using an RTX 8000. For comparisons against Nvidia's IndeX renderer, NA indicates the presence of cell-centered data, which is currently unsupported.*

data, and a $6.6-15\times$ improvement on the bilinear data sets (Table 4).

We also find that performance improvements for our general unstructured elements are much more significant than our performance improvements for tetrahedra. This more substantial improvement makes sense. The baseline of our reference method is much lower for general unstructured meshes than for tetrahedral meshes, and thus there is much more room for improvement. In particular, when data sets consist of bilinear elements with per cell values, the reference method requires a full Newton-Raphson optimization to determine if the point lies within a cell. However, with our shared faces method, if we know the elements contain per-cell values, we can avoid this Newton-Raphson optimization altogether, and instead just test if we intersect the bilinear face in the *closest-hit* program before returning the cell value or -1. For unstructured meshes with per-vertex data, our method still requires a full Newton-Raphson optimization to interpolate the field values in the *closest-hit* program. Even so, we still see significant performance improvements for meshes with per-vertex data, on the order of $5.9-6.6\times$ compared to the reference.

Finally, to validate our RT-core accelerated unstructured volume renderer, we measured some performance comparisons against current state of the art in Table 5. From our testing, we are able to demonstrate a clear advantage in terms of unstructured volume rendering performance. Compared to Paraview's OpenGL projected tetrahedra approach, our results see performance improvements between two to three orders of magnitude. However, this projected tetrahedra approach is quite old, and we would not consider it as being state of the art.

More recently updated and more commonly used is the approach by Rathke, which is implemented in Intel's OSPRay renderer. When we measure the performance of OSPRay 2.2.0's unstructured volume renderer against Paraview's projected tetrahedra approach, OSPRay's performance beats Paraview's projected tetrahedra by about an order of magnitude. However, our results again demonstrate another order of magnitude in performance improvements over OSPRay. It is worth noting that the comparison between our approach and OSPRay is more-so due to hardware differences rather than algorithmic differences, since we both implement the approach by Rathke. For our OSPRay comparisons, we chose to compare our high-end workstation RTX 8000 results against an i9-9920x, as this hardware was what we had available during testing; however, in terms of cost, the i9 is significantly less expensive than the RTX 8000. More comparable are the benchmarks of our technique measured on our more budget-comparable RTX 2080 TI, where we still see an order of magnitude of improvement over OSPRay.

Nvidia's IndeX (version 2.4) is arguably the most competitive unstructured volume renderer we compare our performance results against. We consistently outperform IndeX with our approach, although the size of our performance improvements depends significantly on the data set and transfer function. For the *Fusion* data set, IndeX performance roughly matches ours. In this data set, each volumetric point query sample is more expensive; however, due to the selected transfer function, relatively few queries are required before rays reach maximum opacity. As data sets grow larger, we likewise see larger performance improvements compared to IndeX, up to 2× for the hexahedral *Japan* data set. However, our largest performance improvements actually come from the smaller *Jets* data set, where we see a 5.8× performance improvement over IndeX. This dataset contains many elements that are made transparent after the transfer function is applied. It seems the strategy used by IndeX suffers under these cases highly transparent cases, and our point query approach seems to do much better.

Overall, the frame rate of our volumetric raycaster appears to be dependent on the data size, and in absolute terms, decreases as the data size grows. We note that this decrease in frame rate is not due to an increase in cost per-sample but rather due to our relatively naive volume ray marcher. The ray marcher uses a fixed step size and does not implement empty-space skipping or adaptive sampling. Thus for large but sparse data sets, the ray marcher will take a large number of samples in largely homogeneous or empty regions of the data. Although each point query is relatively cheap, in aggregate, they are not. We could likely address this problem by adding support for space skipping or adaptive sampling to our renderer (see, e.g., [17]).

9.4 Power Draw

Finally, it is interesting to compare the different methods in terms of power draw. We log the output of `nvidia-smi` during the rendering benchmarks to monitor power draw. We find that both the `cuda-bvh` and `rtx-bvh` methods always reach roughly the card's maximal power draw (225W for the RTX 2080, 260W for the RTX 2080TI, 280W for the TITAN RTX, 260W for the RTX 8000). However, the RTX triangle-based tetrahedra kernels consistently draw less power, averaging around 170W on the RTX 2080, 250W on the RTX 2080TI, 230W on the TITAN RTX, and 191W on the RTX 8000. By leveraging these new hardware capabilities, our kernels achieve a 2× or higher performance improvement on tetrahedral meshes, while using around 20% less power. We see roughly the same power improvements for general unstructured meshes with cell-centered scalar data, where we achieve a 15× performance improvement. However, general cells with per vertex scalars require a more computationally intense Newton-Raphson operation within the closest program, where our power savings over the reference are more limited.

10 Discussion and Conclusion

Although our results are promising, there are several interesting avenues for future work to explore creative uses of the ray tracing cores.

With regard to addressing our larger goal of exploring wider use of the RT cores, we have successfully shown *one* application where they can be used to accelerate a problem beyond traditional ray tracing. However, more work must be done to extend this initial idea beyond point queries on unstructured meshes. We have now demonstrated how our approach can extend to elements with non-triangular faces, and it is likely that other common unstructured data queries and mesh types could be accelerated as well. For example, *k*-nearest-neighbor and closest point queries are widely used in a broad class of applications, and accelerating such queries would be valuable.

As for the kernels presented in this paper, we believe other applications beyond direct volume rendering could leverage our approach as well. Simulations that combine particle and volumetric data or advect particles through a field could benefit from accelerated point queries. However, such simulations may require a high degree of numerical accuracy. Although the field could be stored in double-precision, RTX supports only single-precision vertex data at this time. Furthermore, such simulations may require higher order polynomial interpolants, whereas our current approach is currently limited to bilinear elements. We believe our technique could be extended to support these higher order elements so long as the surfaces of the mesh elements can be contained within an underestimating and overestimating tessellation. The Newton-Raphson node interpolation and GARP bilinear patch intersection methods we use now could then be replaced with the equivalent higher order techniques.

When it comes to unstructured volume rendering, another caveat with our prototype renderer is that we only explore taking individual samples in a simple ray marcher. However, other techniques based on stepping from element to element may be more efficient or provide higher quality images (e.g., [6, 18, 20]). Some of our general ideas may be applicable to such techniques as well. We note that adding empty-space skipping and adaptive sampling, as suggested by Morrical et al. [17] or Ganter and Manzke [3], would greatly improve the performance and quality of our prototype ray marcher, and would integrate nicely with the presented approach. Beyond volume rendering unstructured meshes specificially, it is likely that the RT cores could be used to accelerate common visualization tasks beyond sampling or space skipping.

Overall, our results are encouraging. Not only did our first attempt to use the RT cores for something beyond classical ray tracing work, each of the three kernels evaluated provided improvements over the software reference, with the fastest methods far exceeding our expectations. On the more challenging bilinear element use case, we have seen even more substantial performance improvements. These results continue to raise new questions for current and future hardware architectures. Beyond point queries, what other problems could we accelerate by leveraging ray tracing hardware? How might changes to future iterations of these or similar hardware units change the answer to that question? Will more general, non-ray tracing use cases shape future hardware designs? We hope our results serve to motivate further investigation into what we believe to be the promising potential of general-purpose ray tracing.

Acknowledgments

The Agulhas data set is courtesy of Dr. Niklas Röber (DKRZ); the Japan Earthquake data set is courtesy of Carsten Burstedde, Omar Ghattas, James R. Martin, Georg Stadler, and Lucas C. Wilcox (ICES, the University of Texas at Austin) and Paul Navrátil and Greg Abram (TACC). Hardware for development and testing was graciously provided by Nvidia Corp. This work is supported in part by NSF: CGV Award: 1314896, NSF:IIP Award: 1602127, NSF:ACI Award: 1649923, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375 and NSF:OAC Award: 1842042.

References

[1] R. T. Biedron, J. R. Carlson, J. M. Derlaga, P. A. Gnoffo, D. P. Hammond, W. T. Jones, B. Kleb, E. M. Lee-Rausch, E. J. Nielsen, M. A. Park, et al. Fun3d manual: 13.6. 2019.

[2] H. Childs. Visit: an end-user tool for visualizing and analyzing very large data. 2012.

[3] D. Ganter and M. Manzke. An analysis of region clustered bvh volume rendering on gpu. In *Computer Graphics Forum*, volume 38, pages 13–21. Wiley Online Library, 2019.

[4] A. S. Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[5] S. Green. Nvidia cloth sample. 2003. `download.nvidia.com/developer/SDK/Individual_Samples/samples.html#glslphysics`.

[6] C. Gribble. Multi-Hit Ray Tracing in DXR. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. 2019.

[7] G. Gu, D. Kim, J. M. Pereira, and R. Raidou. Accurate and memory-efficient gpu ray-casting algorithm for volume rendering unstructured grid data. In *EuroVis (Posters)*, pages 77–79, 2019.

[8] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed Up Mixed-precision Iterative Refinement Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (Proceedings of Supercomputing '18)*, 2018.

[9] E. Haines and T. Akenine-Möller. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Springer, 2019.

[10] M. P. Howard, J. A. Anderson, A. Nikoubashman, S. C. Glotzer, and A. Z. Panagiotopoulos. Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units. *Computer Physics Communications*, 203:45–52, 2016.

[11] T. Kim and M. C. Lin. Visual simulation of ice crystal growth. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 86–97. Eurographics Association, 2003.

[12] A. Knoll, R. K. Morley, I. Wald, N. Leaf, and P. Messmer. Efficient particle volume splatting in a ray tracer. In *Ray Tracing Gems*, pages 533–541. Springer, 2019.

[13] J. H. Krüger and R. Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. In *Computer Graphics Forum*, volume 24, pages 685–694. Amsterdam: North Holland, 1982-, 2005.

[14] E. S. Larsen and D. McAllister. Fast Matrix Multiplies Using Graphics Hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, 2001.

[15] A. Maximo, R. Marroquim, and R. Farias. Hardware-assisted projected tetrahedra. In *Proceedings of EuroVis'10*, 2010.

[16] K. Moreland and E. Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119. Eurographics Association, 2003.

[17] N. Morrical, W. Usher, I. Wald, and V. Pascucci. Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. October 2019.

[18] P. Muigg, M. Hadwiger, H. Doleisch, and E. Groller. Interactive Volume Visualization of General Polyhedral Grids. *IEEE Transactions on Visualization and Computer Graphics*, 2011.

[19] P. Muigg, M. Hadwiger, H. Doleisch, and H. Hauser. Scalable Hybrid Unstructured and Structured Grid Raycasting. *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[20] B. Nelson, E. Liu, R. M. Kirby, and R. Haimes. ElVis: A System for the Accurate and Interactive Visualization of High-Order Finite Element Solutions. *IEEE Transactions on Visualization and Computer Graphics*, 2012.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[22] NVIDIA. NVIDIA OptiX 6.0–Programming Guide. `https://bit.ly/2ErCDti`, 2018.

[23] NVIDIA. NVIDIA Turing GPU Architecture. `https://bit.ly/2NGLr5t`, 2018.

[24] J. Olliff. Efficient adjacency queries and dynamic refinement for meshfree methods with applications to explicit fracture modeling. 2018.

[25] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[26] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, and A. Robison. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 2010.

[27] J. Pineda. A parallel algorithm for polygon rasterization. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 17–20, 1988.

[28] B. Rathke, I. Wald, K. Chiu, and C. Brownlee. SIMD Parallel Ray Tracing of Homogeneous Polyhedral Grids. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2015.

[29] A. Reshetov. Cool patches: A geometric approach to ray/bilinear patch intersections. In *Ray Tracing Gems*, pages 95–109. Springer, 2019.

[30] M. Rumpf and R. Strzodka. Level Set Segmentation in Graphics Hardware. In *Proceedings of the 2001 International Conference on Image Processing*, 2001.

[31] M. Rumpf and R. Strzodka. Using Graphics Cards for Quantized FEM Computations. In *Proceedings of the VIIP Conference on Visualization and Image Processing*, 2001.

[32] J. L. Salmon and S. M. Smith. Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC. 2019.

[33] R. Sawhney and K. Crane. Monte carlo geometry processing: A grid-free approach to pde-based methods on volumetric domains. *ACM Trans. Graph.*, 39(4), 2020.

[34] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. In *Proceedings of the 1990 Workshop on Volume Visualization*, 1990.

[35] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[36] M. Ulmstedt and J. Stålberg. Gpu accelerated ray-tracing for simulating sound propagation in water, 2019.

[37] M. Vinkler, V. Havran, and J. Bittner. Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing. In *Computer Graphics Forum*, volume 35, pages 68–79. Wiley Online Library, 2016.

[38] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navrátil. OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2017.

[39] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. In *Proceedings of High Performance Graphics*, 2019.

[40] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree - A

Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics*, 2014.

[41] I. Wald, S. Zellmann, W. Usher, N. Morrical, U. Lang, and V. Pascucci. Ray tracing structured amr data using exabricks. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[42] B. Yang and T. A. Laursen. A contact searching algorithm including bounding volume trees applied to finite sliding mortar formulations. *Computational Mechanics*, 41(2):189–205, 2008.

[43] S. Zellmann, M. Weier, and I. Wald. Accelerating force-directed graph drawing with rt cores. In *IEEE Visualization (Short Papers)*, 2020. arXiv:2008.11235.

Nate Morrical is a PhD student at the University of Utah, and is currently working under Valerio Pascucci as a member of the CEDMAV group in the Scientific Computing and Imaging Institute (SCI). His research interests include high performance GPU computing, real time ray tracing, and human computer interaction. Prior to joining SCI, Nate received his B.S. in Computer Science from Idaho State University, where he researched interactive computer graphics and computational geometry under Dr. John Edwards.

Ingo Wald is a Director, Ray Tracing at NVIDIA. He got his master's degree from Kaiserslautern University, and a PhD from Saarland University (both on ray tracing related topics); and after that served as a Post-Doc at the MPI Saarbrücken, as a Research Professor at the University of Utah, and as Tech Lead for Intel's software-defined rendering activities (in particular, Embree and OSPRay). Ingo has co-authored more than 75 papers, multiple patents, and several widely used software projects around ray tracing; his interest still revolve around all aspects of efficient and high-performance ray tracing, from visualization to production rendering, from real-time to off-line rendering, from hard- to software, etc.

Will Usher is a Graduate Research Assistant at the Scientific Computing and Imaging Institute at the University of Utah, working with Valerio Pascucci. Before joining the Ph.D. program at Utah Will obtained a B.S. in physics with a minor in computer science at the University of California, Riverside in 2014. His research interests cover a range of areas in scientific visualization and computer graphics including: distributed rendering, virtual reality, in situ visualization and ray tracing.

Valerio Pascucci is the founding Director of the Center for Extreme Data Management Analysis and Visualization (CEDMAV) of the University of Utah. Valerio is also a Faculty of the Scientific Computing and Imaging Institute, a Professor of the School of Computing, University of Utah, and a Laboratory Fellow, of PNNL. Before joining the University of Utah, Valerio was the Data Analysis Group Leader of the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory, and Adjunct Professor of Computer Science at the University of California Davis. Valerio's research interests include Big Data management and analytics, progressive multi-resolution techniques in scientific visualization, discrete topology, geometric compression, computer graphics, computational geometry, geometric programming, and solid modeling. Valerio is the coauthor of more than one hundred refereed journal and conference papers and has been an Associate Editor of the IEEE Transactions on Visualization and Computer Graphics.