Manual tuning of physics-based animation parameters to explore a simulation outcome space or achieve desired motion outcomes can be notoriously tedious. This problem has motivated many sophisticated and specialized optimization-based methods for fine-grained (keyframe) control, each of which are typically limited to specific animation phenomena, usually complicated, and, unfortunately, not widely used.

In this paper, we propose **Unified Many-Worlds Browsing (UMWB)**, a practical method for sample-level control and exploration of physics-based animations. Our approach supports browsing of large simulation ensembles of arbitrary animation phenomena by using a unified volumetric WorldPack representation based on spatiotemporally compressed voxel data associated with geometric occupancy and other low-fidelity animation state. Beyond memory reduction, the WorldPack representation also enables unified query support for interactive browsing: it provides fast evaluation of approximate spatiotemporal queries, such as occupancy tests that find ensemble samples (“worlds”) where material is either IN or NOT IN a user-specified spatiotemporal region. WorldPacks also support real-time hardware-accelerated voxel rendering by exploiting the spatially hierarchical and temporal RLE raster data structure. Our UMWB implementation supports interactive browsing (and offline refinement) of ensembles containing thousands of simulation samples, and fast spatiotemporal queries and ranking. We show UMWB results using a wide variety of physics-based animation phenomena—not just Jell-O®.

**CCS Concepts:** • Computing methodologies → Animation; Physical simulation; Volumetric models.

Additional Key Words and Phrases: Motion control, interactive control, browsing, ranking, simulation ensembles, volume data, compression, RLE, WorldPack, ray tracing Jell-O®.

**ACM Reference Format:**


1 INTRODUCTION

Tuning parameters of physically based animations is a notoriously tedious task required to either explore a motion outcome space or achieve desired behaviors, such as “what goes where” scenarios, e.g., tossing pieces of Jell-O® into a bowl. Manual exploration of the
In this paper, we propose **Unified Many-Worlds Browsing (UMWB)** for sample-level control and exploration of arbitrary physics-based animations—a reimagining of Twigg and James [2007] for rigid-body animations. Please see Figure 1 for a preview of our results, and Figure 2 for an overview of our technique. Our approach begins by sampling (and optionally refining) animations in parallel from a distribution of possible input parameters (initial conditions, geometric and design variations, material properties, control forces, stochasticity, etc.) to create an animation ensemble. Simulation summarization is performed to support unified browsing of arbitrary animations. We volumetrically rasterize and quantize 3D animation state, such as geometry and fields of interest, to create binary occupancy and low-fidelity attribute value grids. These fields are further compressed using temporal run-length encoding (RLE), followed by spatially hierarchical decorrelation, and then stored in a compressed format, a **WorldPack**, for unified browsing. For further scalability, the original high-fidelity simulations can be discarded—most are never used—then recomputed later, if needed, using their sample parameters, thereby resulting in significant storage reductions.

The compressed volumetric **WorldPack** sample representation provides just enough information to summarize the animation, while also enabling fast unified query support for interactive browsing. Given user-specified spatiotemporal regions, e.g., a mouse-drawn bounding box, we can perform fast spatiotemporal occupancy tests to find ensemble samples (“worlds”) where material is either IN or NOT IN a specific spacetime region, and thus enable “what goes where” animation queries. The compressed representation enables fast evaluation of general query and ranking predicates, such as spatiotemporal integrals.

We further leverage **WorldPack**’s spatially hierarchical and temporal RLE raster representation to perform hardware-accelerated ray tracing. Although of a greatly reduced visual fidelity compared to production quality animations, our system can display hundreds of multi-physics simulation summaries at interactive rates with modest in-core memory footprints.

Finally, we demonstrate our prototype UMWB implementation with physics-based animation content generated by a wide variety of phenomena and simulators. Our system can support interactive browsing (and offline refinement) of ensembles containing...
thousands of simulation samples, efficiently perform spatiotemporal queries at interactive rates, and rank and display selections interactively in a browser for easy use by animators. Our browser demonstration and data are available at https://github.com/stanford-gfx/umwb.

2 RELATED WORK

Tuning Parameters and Design Spaces. A fundamental problem in physics-based animation is the tuning of many simulation and control parameters. Manual exploration of parameter spaces is often done using tedious online interactive tuning, or offline brute-force (parallel) sampling of so-called parameter “wedges” (e.g., in SideFX Houdini [SideFX 2021]) or animation ensembles (c.f. “Design Galleries” [Marks et al. 1997]). Our query-driven exploration scheme is inspired by a larger body of work on user-guided exploration of high-dimensional, highly varied design spaces. Some design spaces can be sampled in real time: the solver or model can instantly produce an outcome from a set of input parameters, in such domains as parametric BRDFs [Talton et al. 2009], color grading of photographs [Koyama et al. 2020], and 2D particle systems [Shimizu et al. 2020]. Fast solvers are conducive to a fast, interactive user exploration experience. In order to support interactive browsing when solvers do not run at interactive rates, such as in this paper, prior work relies on computing an ensemble of simulation samples in an offline step, such as work on Design Galleries [Marks et al. 1997]. This pipeline was applied to fire and smoke volumes at low resolutions [Bruckner et al. 2020], interactive spacetime control using a prototype interface for keyframe control of physics-based simulations led to several advancements [Witkin and Kass 1988] expressed keyframe control as the solution to a constrained spacetime optimization problem, and explored how to programatically imbue physics-based animations with anticipation, follow-through, squash-and-stretch, and timing [Lasseter 1987]. Later work explored genetic algorithms [Tang et al. 1995], interactive spacetime control using a prototype interface for sketching motion edits [Cohen 1992], and entire rigid-body motions using multiple-shooting optimization [Popović et al. 2003].

Rigid-body Animation Control. Many methods for control and steering have emerged in response to the often unintuitive, finicky, and time-consuming process of tuning rigid-body animation parameters to produce a desired outcome. These techniques introduce algorithms that work in tandem with users, solving, searching, or optimizing for a set of initial conditions that produce a user-specified desired simulation end state or outcome. The dream of fine-level keyframe control of physics-based simulations led to several advances. Witkin and Kass [1988] expressed keyframe control as the solution to a constrained spacetime optimization problem, and explored how to programatically imbue physics-based animations with anticipation, follow-through, squash-and-stretch, and timing [Lasseter 1987]. Later work explored genetic algorithms [Tang et al. 1995], interactive spacetime control using a prototype interface for sketching motion edits [Cohen 1992], and entire rigid-body motions using multiple-shooting optimization [Popović et al. 2003].

Idealized simulation models are commonly enriched using (stochastic) parameter variations, e.g., perturbed contact normals, to produce more varied yet visually plausible dynamics by virtue of realistic statistical variations [Barzel et al. 1996] and the limitations of human perception [O’Sullivan et al. 2003]. Producing richer animation ensembles enables opportunities for sample-level control wherein animation control is achieved by selecting desired outcomes from available ensemble samples, for example, to produce a plausible yet desirable pool break in virtual billiards [Barzel et al. 1996], or to impose spatiotemporal object constraints or steer ensemble refinement [Twigg and James 2007].

Beyond Rigid Bodies. There are several works that have explored simulation control and specialized guiding techniques for domains other than rigid bodies. Examples include fluid animation controllers [Foster and Metaxas 1997] and keyframe-based smoke control [Fattal and Lischinski 2004; Treuille et al. 2003]. Adjoint control methods assist with many control parameters for fluids [McNamara et al. 2004] and particle systems [ Wojtan et al. 2005]. Other methods for fluids include guide-based techniques [Sato et al. 2021; Shi and Yu 2005]; neural style transfer [Guo et al. 2022; Kim et al. 2019] for indirect control; strategically placed control forces and interactive local edits [Pan et al. 2013; Schoentgen et al. 2020; Thürey et al. 2006]; methods for stylizing fluids with neural networks [Kim et al. 2020; Yan et al. 2020]. Unfortunately methods for calculating control forces to carefully guide simulations are domain-specific and, for instance, formulations tuned to controlling smoke do not trivially generalize to controlling rigid bodies or elastic objects. For example, specialized methods exist for coarse-to-fine control of thin shells using moment-based constraints [Bergou et al. 2007].

Control techniques for multiphysics simulations that can generalize to several physical models are less common. Some attempts include controllers for reduced-order dynamical systems [Barbić et al. 2020; Yan et al. 2020]. Unfortunately methods for calculating control forces to carefully guide simulations are domain-specific and, for instance, formulations tuned to controlling smoke do not trivially generalize to controlling rigid bodies or elastic objects. For example, specialized methods exist for coarse-to-fine control of thin shells using moment-based constraints [Bergou et al. 2007].
and Popović 2008]. Recently, Ma et al. [2018] trained a controller using deep reinforcement learning for coupled fluids and rigid bodies, but only offers control on the fluid in the domain.

**Volumetric Data Structures.** Our work on browsing-specific compression formats is closely related to several previous papers on data structures optimized for storing large, sparse volumes. Sparse regular grids are common in computer animation applications, for representing everything from scenes to vector fields. As a result, a great number of papers explore different ways to handle common issues with these grids, such as high memory footprints and expensive data access, including VDB [Museth 2013], DT-grid [Nielsen and Museth 2006], SPGrid [Setaluri et al. 2014], and DB+grid [Museth 2011]. Several methods are inspired by volumetric octrees [Samet 2006] for exploiting spatial sparsity, such as sparse paged grids [Aanjaneeya et al. 2017] and wide-branching tile trees [Nielsen et al. 2018]. However, most of these data structures target dynamic data using instantaneous, per-frame representations; in contrast, our work targets browsing of precomputed simulation ensembles where simulation samples are not modified at browsing time. In this context, NanoVDB [Museth 2021], which stores immutable simulation snapshots with static topology, is more similar to our WorldPack representation. One key difference is that NanoVDB and its predecessors store a separate serialized data structure for each timestep, independent of data at other timesteps. However, many simulations contain both temporal and spatial sparsity and redundancy, especially when quantized heavily. Therefore, rather than storing a data structure per-timestep, our proposed WorldPack data structure leverages both in-space and in-time compression to further reduce the memory footprint and accelerate queries.

Run-length encoding (RLE), our temporal compression strategy, has been widely applied to volume data in previous work, particularly for representing sparse level sets [Houston et al. 2006, 2004], and signed distance functions [Curless and Levoy 1996]. But these methods use RLE solely for in-space compression; to our knowledge, applying RLE compression in-time on volume data is a largely unexplored direction, in part because of the pervasive nature of per-frame animation processing. Coupled in-time and in-space compression appears in common 2D video compression schemes like GIF89a and H.264 [Sayood 2017], and can effectively reduce memory costs while preserving quality. In contrast, our data structure is designed specifically for fast processing of spatiotemporal user queries on low-fidelity volume animation data.

Finally, our WorldPack queries exploit the fact that vector sets compressed using RLE (or PackBits) can be used to accelerate vector computations (dot products, summation, etc.) [Oyamada et al. 2018].

**Animation Compression.** Many-Worlds Browsing used rigid-body trajectory compression to support larger in-core ensembles [Twigg and James 2007], and the center-of-mass trajectory for fast, approximate spatial queries. Rigid-motion compression rates are good for ballistic trajectories, but degrade for objects undergoing many complex collisions (c.f. [Jeruzalski et al. 2018]). In computer animation, compression schemes exist for various animated content, such as deforming mesh animations [Lengyel 1999; Sattler et al. 2005], character animation databases [Arikan 2006], fluid animation fields using DCT-based compression [Jones et al. 2016], etc., however they tend to be domain-specific and not easy to integrate with approximate browsing queries on arbitrary animated content. In contrast, we exploit a unified low-fidelity representation to achieve compression, fast rendering and approximate queries of arbitrary animations during ensemble browsing. In other fields, the use of compressed representations of data that allows for answering queries about that data is referred to as “sketching” [Nelson 2011]. Lastly, low-fidelity simulations for browsing are orthogonally related to the use of adaptive precision simulation [Hu et al. 2021; Yeh et al. 2009].

### 3 SUMMARIZING SIMULATIONS WITH WORLDPACK

#### 3.1 Design Goals

**3.1.1 Simulation Summaries.** Unfortunately, storing ensembles in core for on-the-fly query evaluation can quickly become intractable given the sheer amount of simulation data often dumped. For example, a single smoke animation sample can contain several gigabytes of simulation data, and we anticipate ensemble sizes on the order of hundreds to thousands of samples. Furthermore, it is impractical to use raw simulation ensemble data to support interactive query performance and the visualization of selected results.

To this end, we require a data structure that can summarize simulation data in support of the following UMBW design desiderata:

1. **Arbitrary & Unified:** We desire support for arbitrary physics-based animation models (solids, fluids, etc.) by using an approximate but unified data representation.
2. **Expressive:** We desire sufficient spacetime resolution to express spatiotemporal animation queries of practical interest.
3. **Low Memory:** We desire in-core query processing and thus require a low-memory footprint for the entire summarized animation ensemble.
4. **Fast Queries and Ranking:** The data structure should be optimized for interactive evaluation of spatiotemporal queries.
5. **Fast Rendering:** Since the original simulation data is unavailable, the summary must itself provide a fast visual substitute.

Design decisions made under these tenets resulted in WorldPack, a data structure for interactive exploration of large simulation ensembles. With WorldPack, users may interact with rasterized ensemble data at interactive rates in order to explore hundreds to thousands of ensemble samples and narrow down the outcome space to only the samples they find interesting.

**3.1.2 Simulation Data Quantization.** Raw simulation data dumped from solvers are often at the high precisions required for per-timestep solves. We emphasize that the intention of the WorldPack data structure is to store an expressive but lightweight sketch of an animation over time; therefore, all simulation data that we store in WorldPack will be quantized to lower precision.

**Spacetime Occupancy:** For what-goes-where queries that form the bulk of our user interaction model, just knowing if a spatiotemporal region contains an object or not is generally sufficient to sift through ensemble samples. We consider this a question of *spacetime occupancy* in which simulated material of interest is identified by the user using an indicator function $I_{\Omega} : (x, t) \rightarrow \{0, 1\}$. Specifically, let the object material occupy the spacetime region $\Omega \subset \mathbb{R}^3 \times \mathbb{R}_+$. 


While we lose accuracy for these values, we gain the flexibility to explore simulation spatial data in a query predicate. Notably, data reduction operators (e.g., repeatedly and efficiently applying data reduction operators such as min, max) to simulation data region $R$, to check the scalar $v$ attribute value itself, we instead store an integer index $i_v$.

Users may chain together queries to explore arbitrary physics-based ensembles in a simple and unified manner. From our definitions, it follows that evaluating if queries are true or false for a given ensemble sample amounts to processing spatiotemporal operators, e.g., repeatedly and efficiently applying data reduction operators (such as min, max) to simulation data region $R$, to check the scalar in a query predicate.

### 3.2 WorldPack Concepts

**3.2.1 Exploiting Spacetime Structure.** While simulation spatial dimensions can grow to intractable sizes, we generally do not need to store data for every single spacetime point in the domain. First, we observe that for many artist animations, quantized simulation data contains a large amount of spatiotemporal redundancy: values in a given spatial neighborhood are often relatively consistent over portions of time—an extreme case of this redundancy is with the rasterized, boolean (0 or 1) occupancy indicator function where changes between frames only occur on the object’s boundary. Second, relevant data exhibits sparsity, as it is usually constrained to compact regions within the domain. This sparsity may manifest itself as empty space (“density is 0 in region $R$”) interspersed with static geometry (“occupancy exists in region $R$”). Many popular data structures like VDB [Museth 2013] exploit the spatial sparsity of simulation domains for better compression and efficient sampling. However, these data structures are generally stored as snapshots for per-frame processing, e.g., an artist may dump a sparse volume for each frame of an animated sequence. While popular data representations can vastly improve memory footprints for a given timestep, storing per-timestep volume representations over hundreds or thousands of timesteps can quickly get out of hand. It is not uncommon for a cache of per-timestep sparse volumes for a single animated sequence to demand gigabytes of memory.

Because sparse volumes are customarily compressed using strategies that leverage spatial redundancy and dumped per-timestep, they fail to exploit spatiotemporal redundancy. We observe that simulation data is often both consistent in space and time. To reference our previous example, spatiotemporal redundancy can appear as “density is 0 within region $R$ for a span of $T$ timesteps.” Utilizing redundancy in both space and time is a largely unexplored space, but in order to store ensembles with sizes on the order of thousands of samples in core, we must look further than compression with just a spatial hierarchy.

### 3.3 WorldPack Construction

**3.3.1 Quantization.** As discussed earlier, we start by quantizing simulation data to lower-precision values in a field-type-specific manner. For spacetime occupancy, 1-bit quantization is already performed by the indicator function, $I_R(x, t)$.

For a scalar attribute value, $v \in \mathbb{R}$, we discretize the continuous interval $[\min v, \max v]$ into $b_v$ bins, where $\max v$ (or $\min v$) are the maximum (or minimum) value that the $v$ attribute ever takes for a simulation sample. Let the length $L = \max v - \min v$ be divided into $b_v$ distinct bins. Every attribute value $v$ thus falls into a value bin with integer index $i_v \in [0, b_v)$. Rather than storing the higher-precision $v$ attribute value itself, we instead store $i_v$.

For all following steps, we act on quantized simulation data using ordinal values, e.g., 8-bit value-bin indices, rather than the raw values.

**3.3.2 Rasterization.** In order to support unified data exploration for arbitrary simulations, regardless of solver-specific representations (e.g., particles, grids, meshes, etc.), we rasterize per-sample, per-timestep simulation data onto coarse, uniform voxel grids, with user-specific cell size, $h$. All relevant attribute values (e.g., occupancy, temperature, density, etc.) are rasterized onto separate uniform grids. A simulation domain that has a spatial height, width, and depth of $H$, $W$, and $D$, respectively, is then converted into a 3D uniform voxel...
grid of dimensions $N_x \times N_y \times N_z = [H/h] \times [W/h] \times [D/h]$. We refer to the $ijk^{th}$ cubical cell as a raster block, and denote its spacetime region as $\Omega_{ijk} \subset \mathbb{R}^3 \times [0, T_{\text{max}}]$. The projection of simulation data to a single raster block value can be done via standard reductions. For example, spacetime occupancy is rasterized by approximating a conservative max reduction of the indicator function:

$$I_{ijk} \equiv \max_{(x,t) \in \Omega_{ijk}} I_{\Omega}(x,t). \quad (2)$$

Discussion. Rasters only need to be coarse summaries of simulation data, so for browsing, a large $h$ is preferable and enables efficient inspection of simulation-data trends across coarse spacetime chunks. Additionally, because WorldPacks are built as summaries of already-dumped simulation data, the finer resolutions required for simulation computations or high-quality rendering are unnecessary.

3.3.3 Temporal Compression. Let us consider a single ensemble sample at a time. For every timestep in this sample $S$, we have rasterized relevant attribute values into summaries composed of raster blocks. Because quantized simulation state is usually temporally redundant, we observe that, for a given attribute value, a raster block can have the same or similar values for significant stretches of time. Therefore, we use run-length encoding (RLE) of the raster block’s values in time for the length of the simulation. RLE converts repeated data values into a sequence of $<\text{count}, \text{value}>$ pairs, e.g., given the sequence of values: $\{2, 2, 0, 4, 4, 4, 4, 4, 4, 4, 4\}$, RLE would produce the following $<\text{count}, \text{value}>$ tuples, commonly referred to as runs: $<3, 2>, <1, 0>, <8, 4>$. Notice how RLE has exploited the sequence’s temporal redundancy, to reduce the number of values being stored from 12 to 6.

While strictly-repeating values are rare in floating-point simulation data, quantization creates redundancy in sequences that are very close in value. As a result, temporally encoded sequences usually have a smaller memory footprint in practice. Processing RLE-encoded sequences when evaluating temporal integrals is also often faster than storing a series of individual values because there is usually a) less to process, and b) processing does not require decoding the entire sequence (c.f. [Oyamada et al. 2018]). We discuss specifics of RLE-in-time operations in §4.

3.3.4 Spatial Compression. Next, we exploit the spatial consistency of simulation data: spatiotemporal neighborhoods often exhibit similar behavior. So, we build a shallow, wide octree on the RLE-in-time signals. Starting from the root, we only subdivide a node if the RLE sequences contained in the node’s raster blocks vary. Otherwise, storing a single RLE sequence for the node’s region is sufficient. The maximum depth for a hierarchy $D_{\text{e}}$ chosen to be a small constant; because all RLE-in-time sequence data is stored at leaf nodes, a shallow structure allows us to traverse from root to leaf in a bounded number of steps.

Node Representation. Because octrees are regular hierarchies, we only need to store the dimensions and position of the root node. After that, as long as we store node children in a fixed order, the location and size of internal and leaf nodes can be inferred from depth [Samet 2006].

At every node, we store a flag indicating if the node has children or not; a flag value of true indicates that a node has 8 children, while a flag value of false indicates that a node has 0 children. Nodes store RLE-in-time data differently depending on one of three cases, as in Figure 7:

- Internal nodes do not store RLE-in-time sequences, and only need to store a reference (either through pointers or memory...
3.3.5 Linear Representation. A final, optional step is to linearize the data structure. Because the data structure is static and read-only, we replace all pointers between parent and children nodes with 32-bit memory offsets. Linearized WorldPacks are stored in breadth-first order in a contiguous, unbroken block of memory.

Linearized WorldPacks can be transferred rapidly from disk to memory, and from CPU to GPU. The latter enables accelerated processing of computationally intensive algorithms like volume rendering, which is relevant for visualizing WorldPacks in our browser (see §4).

4 ACCELERATED BROWSING FUNCTIONALITY WITH WORLDPACKS

4.1 Spatiotemporal AABB Queries

Users may define query regions as AABB spacetime bounding boxes, after which the octree-based spatial hierarchy is used to perform efficient processing of WorldPack-AABB intersections. Processing a query uses three steps described in §4.1.1-§4.1.3 (see Figure 8).

Fig. 6. RLE-in-time Compression: WorldPack utilizes RLE-in-time to reduce the behavior in a single raster block over the course of hundreds of timesteps. Following this step, each raster block will have a single RLE encoded signal. (Top Left) Some attribute values are represented in simulators with higher-precision floating point numbers, which may be close in value but do not strictly repeat. (Top Right) Similarly to how we coarsened value but do not strictly repeat. (Top Right) Similarly to how we coarsened time and value fall into the same value bin. Finally, we RLE-in-time encode lower-precision bin IDs rather than higher-precision floating point values (Bottom) into <count, value> pairs. In the case of geometric occupancy, we use just two bins: occupancy either exists, or does not exist, within a raster block. For attribute values requiring finer-grained storage like angles or color, we use 256 bins.

Fig. 7. WorldPack Structure: We show a WorldPack constructed for geometric occupancy on a domain with one spatial dimension, \(x\), and a temporal dimension \(t\). More complex domains are too large to be shown here, but even this toy example contains similar features to our examples: large areas of spatiotemporal redundancy interspersed with areas of distinctness. (Top) Both axes are partitioned into spacetime raster blocks; raster blocks occupied at \((x, t)\) are shown in blue. Every raster block has an associated RLE-in-time sequence. (Bottom) We build a shallow, wide octree on the raster blocks. If any raster blocks within an internal octree node \(O_{\text{internal}}\) have different RLE-in-time sequences, we split \(O\) into child octrees: these internal nodes are shown in purple. If \(O\) is at the hierarchy’s \(O_{\text{depth max}}\) and has any spatiotemporally distinct raster blocks, \(O\) is a leaf, and stores an RLE-in-time sequence for each of its raster blocks: such nodes \(O\) are shown in yellow. If \(O\) is not at the hierarchy’s \(O_{\text{depth max}}\) and all raster blocks in \(O\) have the same RLE-in-time sequence, then \(O\) does not branch and only stores a single RLE-in-time sequence for the entire region: such nodes \(O_{\text{depth max}}\) are shown in green.
4.1.2 **AABB/RLE-in-time Intersection (Step 2/3).** Once we have the leaf octants and raster blocks that are completely encapsulated by the spatial bounds, we need to find which parts of their full RLE-in-time sequences fall into the AABB region’s temporal interval $T_{\text{query}}$. Every RLE-in-time $<\text{count}, \text{value}>$ unit constitutes a single interval with length $\text{count}$. An RLE-in-time sequence is thus a series of intervals that, by construction, do not overlap with each other. We can treat the operation as a search for overlapping ranges between $T_{\text{query}}$ and the RLE-in-time interval sequence and apply a 1D sweeping method. We might find overlapping ranges faster with, e.g., a non-overlapping interval tree, though, in practice, more advanced structures are unnecessary because the number of $<\text{count}, \text{value}>$ units composing our RLE-in-time sequences is quite small; we report the average number of $<\text{count}, \text{value}>$ units per raster block for WorldPacks storing geometric occupancy in Figure 9.

4.1.3 **Evaluate Predicate (Step 3/3).** Once we have determined which RLE-in-time runs overlap with $T_{\text{query}}$, we process their values based on the query’s predicate. These operations are conducted directly on the RLE-in-time compressed representation, without need for decompression: an RLE-in-time sequence implies that a value occurs at a raster block over $\text{count}$ consecutive timesteps. For example, if the predicate asks to sum all values in spatiotemporal region $R_{\text{query}}$, perhaps for later ranking of ensemble samples by total density in $R_{\text{query}}$, we aggregate the $(\text{count} \times V_i)$ products for all RLE-in-time runs within $R_{\text{query}}$:

$$\text{sum}_{\text{block}} = \sum_{i=0}^{\ell_{\text{block}}} n_i V_i$$  \hspace{1cm} (3)$$

where $\ell_{\text{block}}$ is the length of the RLE-in-time sequence for a given raster block, $n_i$ is the count of the $i^{\text{th}}$ RLE unit, and $V_i$ is the value of the $i^{\text{th}}$ RLE unit. If the predicate instead performs an IN query for
spatiotemporal region $R_{\text{query}}$, we check for RLE-in-time sequences in $R_{\text{query}}$ with value 1, the single-bit indicator for occupancy:
\begin{align}
\text{IN} : \quad & \max_{0 \leq i < \ell_{\text{block}}} V_i > 0, \quad (4) \\
\text{NOT IN} : \quad & \max_{0 \leq i < \ell_{\text{block}}} V_i = 0. \quad (5)
\end{align}

Early exits are used to speedup query evaluation. We refer readers to [Oyamada et al. 2018] and [Ding-tao et al. 2008] for a more detailed discussion of accelerated primitive operations on RLE sequences. Like Oyamada et al. [2018], we find that operating on RLE sequences is usually faster than iterating and processing individual values at every timestep; we show an ablation in Figure 10.

The overall speedup of query evaluation using our WORLDPack data structure and WORLDPack-AABB intersection algorithm is significant; most IN and NOT IN queries across region sizes, positions, and ensembles are processed within $10^{-3}$ seconds.

![Query Processing Times Across Data Structures](image)

**Fig. 10. Query Processing** We measure the processing time for 64 IN queries on a sample from our Jell-O ensemble. Query spatial regions are all placed at the center of domain, and have increasing region spatial volume, shown on the x-axis. We evaluate these queries using three data structures: (a) WORLDPack (blue), (b) a uniform RLE-in-time grid (orange) which uses RLE-in-time compression but no spatial hierarchy, and (c) raw data (green), which uses a spacetime uniform grid but no spatial or temporal compression. We measure time in seconds and plot a log-scale. As these query regions increase in size, WORLDPack's keep a steady processing time due to their spatial hierarchy. The dip in processing times at a query volume of 0.4 is due to early exit: as the query regions grow, they eventually encapsulate some new geometric occupancy that allows us to terminate evaluation earlier.

### 4.2 WORLDPack Rendering

As part of the browsing interface, users may explore 3D simulation samples via a navigable, rendered interface with the ability to pan, zoom, and rotate the camera around the scene. Users may compose and visualize spatiotemporal queries from within this scene.

To avoid dumping large amounts of per-timestep scene geometry for visualization, we chose to render the WORLDPack data structure from a fragment shader using a simple ray tracer. WORLDPack's in-time compression allows us to render the scene at any timestep from a single dumped file, enabling users to flexibly navigate the scene without having to extricate, dump, and store extraneous per-timestep simulation geometry. As we will discuss, WORLDPack's spatial hierarchy, spatial sparsification, and support of low-precision attribute value storage are particularly beneficial for GPU ray casting and shading.

#### 4.2.1 Attribute Value Encoding

For basic material shading, we will need to know for $I_2(x, t)$ (a) whether $I_2(x, t) = 0$ (contains geometric occupancy) and if so, (b) the diffuse color $color_2(x, t)$ and (c) the normal $N_2(x, t)$ of the object at $I_2(x, t)$. We need to export this data per raster tile in a low-memory, low-precision format for WORLDPack to leverage spatiotemporal redundancy, while keeping just enough information to enable rendering with these values.

We make several simplifications to reduce the amount of data that needs to be WORLDPack'd. First, rather than storing a 32-bit RGBA color for $color_2(x, t)$, we instead rasterize an 8-bit color ID, which we use to lookup in, e.g., a hash table, the full RGBA color value. While this assumption gives us an upper bound of 255 separate materials in the scene, our examples (see §5) do not strain this limit. We parameterize $N_2(x, t)$ with two polar coordinates $\theta$ and $\phi$. Both polar angles are binned into $0$–$255$ buckets of size $2\pi/255$ radians, for a total of a 16 bits per normal.

We thus reduce shading information (occupancy, color, and normals) per-raster-block into a 24-bit unit for a given timestep. We pack these lower-precision units into our WORLDPack format.

#### 4.2.2 Ray casting

Our WORLDPack rendering algorithm is similar to the volumetric rendering pipeline described by [Laine and Karras 2010], which similarly exploits the regular topology of spatial octrees for efficient ray casting. We therefore refer readers to [Laine and Karras 2010] for an in-depth description of ray casting voxel octrees on the GPU; in summary, we perform ray casts into the spatial hierarchy and incrementally traverse leaf nodes intersected by the ray in depth-first order. Once a ray hits geometry, we lookup shading attributes as inputs to a simple Phong shading model, and the traversal is complete.

The key difference between the GPU-accelerated ray casts described in [Laine and Karras 2010] and ours is the encoding of surface geometry. Laine and Karras [2010] use a timestep-specific octree-simulation data transferred to the GPU is from a single snapshot in time (see Figure 11) so leaf-node occupancy tests can be performed with a single lookup. In contrast, our WORLDPack's use spatially hierarchical RLE-in-time: each leaf node contains an RLE sequence rather than a single occupancy value. So, we must also perform a temporal traversal: simply checking the RLE intervals for occupancy at the time-step that is currently being rendered. While the extra step adds some computation, we can avoid an expensive data transfer of a timestep-specific octree to the GPU for every frame. In our examples, the number of units per RLE-in-time sequence is small, and so the in-time traversal adds only a few extra shader operations; rendering RLE-in-time is negligibly more expensive than rendering a single frame. In Figure 12, we report the average number of $<\text{count, value}>$ units per raster block for WORLDPack's storing shading information quantized in the manner discussed in 4.2.1.
Fig. 11. **Ray Casting WorldPacks**: Rendering WorldPack is reminiscent of rendering a 3D octree. Similar to ray casting a purely spatial hierarchy, a ray is cast from the eye location into the octree and traverses from leaf node to leaf node until it either finds an occupied cell or exits the volume. However, the occupancy test requires decoding the RLE stored at each intersected leaf node to check for occupancy at the current timestep.

Fig. 12. We plot RPF for each WorldPack storing shading information, and plot the average for each ensemble. While the number of runs per raster block is higher in these WorldPacks than they were for WorldPacks storing geometric occupancy, it is still quite small: so, the in-time traversal adds only a few extra GPU shader operations.

Fig. 13. **Smashing Sand Castles**: A sandbox ensemble is created by launching a ball with randomly chosen starting positions and velocities at a sandcastle fashioned out of wet grain. Using exploratory queries we can efficiently say (Left) “give me samples where the front-left turret is smashed by the final timestep,” or (Right) the top turret.

5 RESULTS

We now describe our examples in more detail. Example statistics are shown in Table 1. Please see the accompanying video for all animated results.

**Implementation Details.** Our approach involves multiple components: (1) physics-based simulation, (2) WorldPack exporter, (3) interactive browser with GPU renderer. To support our claim of arbitrary simulation models, we created a diverse assortment of examples using a wide range of dynamics solvers in SideFX Houdini v18.5 (example-specific details are below). We exported WorldPacks straight from Houdini with a Python-based implementation. Our runtime browser is implemented in Javascript and runs via a local server; because they are quite lightweight, an ensemble’s entire set of sample-specific WorldPacks storing geometric occupancy can be stored comfortably in memory. This avoids the need for expensive file IO of WorldPacks when evaluating user queries; WorldPacks can just be loaded once at the start of the browsing session. WorldPacks that store shading information are loaded on an as-needed basis: if the browser needs to display a new simulation sample, we load the sample’s shading WorldPack and send it to the GPU packed into a texture2D. We render WorldPacks with shaders built using WebGL. The browser evaluates user queries at interactive rates, updating the set of feasible samples immediately as the user inserts or drags query regions or adjusts predicates. All feasible samples are shown as representative images within a scrollable menu on the right-hand side of the GUI (see video). Users examine an interactive 3D view of individual WorldPacks by clicking on their representative image. Users can avoid manually examining each feasible sample by ranking using other metrics; we show examples in Figures 14 and 15.

**Smashing Sand Castles (Figure 13).** Our rasterization-based approach supports a wide range of models, including this sand-grain simulation example (simulated in Houdini using 180,000 grains). By applying appropriate queries, the animator can find scenarios where particular turrets are hit or not hit by a thrown object.

**Card Bowling.** WorldPacks can be exported for a single simulated subject, such as lava or Jell-O®, or for several. In this example, we sample an ensemble where a 20-sided die with randomly-sampled initial position and velocity is launched at 10 card “pins” arranged like traditional Candlepin bowling. Dynamics are simulated in Houdini using the Bullet rigid-body solver; each bowling pin is comprised of 585 cards. Queries can be applied to one subject, like the die, or to several in combination, with subject-specific predicates on both the dice and the cards to narrow down the ensemble to find interesting bowling strikes (see Figure 1).
Table 1. **Example Statistics:** For each ensemble, we report the number of ensemble simulation samples, the number of timesteps, the number of input parameters that are randomly varied to create the ensemble. We list the $T_{max}$ chosen for WorldPacks storing geometric occupancy and shading information; we choose $T_{max}$ at a finer resolution for the latter. We also report the average size of each ensemble’s WorldPack in MB. All WorldPacks used for evaluating user queries are $128 \times 128 \times 128$ raster blocks in spatial resolution, with a spatial hierarchy depth of 4. All WorldPacks displayed using hardware-accelerated raycasting in our browser’s interface are $256 \times 256 \times 256$ raster blocks in resolution, with a spatial hierarchy depth of 6. Finally, we report a rounded average serial construction time for constructing a) each WorldPack individually, and b) each entire ensemble; the majority of the time is incurred by solvers, not WorldPack construction. Note that the Staircase ensemble (final row) utilized several rounds of ensemble refinement, so we report statistics for the final stage of refinement, but the total number of samples collected from the entire refinement process.

<table>
<thead>
<tr>
<th>Example</th>
<th>Samples</th>
<th>Num Timesteps</th>
<th>Num Params</th>
<th>$T_{max}$ (Occ.;Shade)</th>
<th>Avg WorldPack Size, MB (Occ.;Shade)</th>
<th>Avg Construction Time (min/sample; min/dataset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sand Castle</td>
<td>104</td>
<td>70</td>
<td>4</td>
<td>3 ; 3</td>
<td>0.089 ; 1.492</td>
<td>25 ; 1040</td>
</tr>
<tr>
<td>Card Bowling</td>
<td>51</td>
<td>155</td>
<td>4</td>
<td>10 ; 5</td>
<td>0.145 ; 2.196</td>
<td>15 ; 758</td>
</tr>
<tr>
<td>Fracture</td>
<td>300</td>
<td>24</td>
<td>1</td>
<td>3 ; 2</td>
<td>0.075 ; 5.526</td>
<td>5 ; 1500</td>
</tr>
<tr>
<td>Jello Rain</td>
<td>46</td>
<td>349</td>
<td>1</td>
<td>10 ; 5</td>
<td>0.321 ; 13.04</td>
<td>20 ; 920</td>
</tr>
<tr>
<td>Smoke</td>
<td>99</td>
<td>164</td>
<td>1</td>
<td>10 ; 10</td>
<td>0.093 ; 2.083</td>
<td>10 ; 990</td>
</tr>
<tr>
<td>Candles</td>
<td>109</td>
<td>70</td>
<td>6</td>
<td>3 ; 3</td>
<td>0.239 ; 4.056</td>
<td>5 ; 543</td>
</tr>
<tr>
<td>Lava Village</td>
<td>72</td>
<td>154</td>
<td>10</td>
<td>10 ; 10</td>
<td>0.103 ; 3.759</td>
<td>20 ; 1440</td>
</tr>
<tr>
<td>Spilt Milk</td>
<td>46</td>
<td>94</td>
<td>3</td>
<td>10 ; 10</td>
<td>0.088 ; 1.871</td>
<td>30 ; 1380</td>
</tr>
<tr>
<td>Staircase</td>
<td>484</td>
<td>92</td>
<td>6</td>
<td>3 ; 3</td>
<td>0.093 ; 1.864</td>
<td>3 ; 1452</td>
</tr>
</tbody>
</table>

**Browsing JELL-O® Brand Gelatin (Figure 1).** We sampled an ensemble of 13 randomly oriented Jell-O cubes falling into a bowl, with each cube simulated as a tetrahedral mesh (5530 tets, 1406 verts.) in Houdini Vellum. Unfortunately some pieces fly out of the bowl. To eliminate simulation samples where Jell-O pieces fell out of the bowl at any time, a user places a large NOT IN query region under the bowl: this constraint allows only samples that do not have geometric occupancy in the selected region.

**Material-Space Browsing of Fracture (Figure 14).** The WorldPack RLE-in-time representation enables us to evaluate spatiotemporal queries over simulation domains at interactive rates. On our largest ensemble, we perform 300 fracture analyses on a heptoroid by subjecting it to random planar impacts, and rendering collected WorldPacks in the body space of the main fragment. By dragging IN and NOT IN queries around both the main object and the shattered detritus, an analyst can quickly sift through the ensemble, and explore different impact responses and degrees of destruction. Metrics calculated on-the-fly from WorldPacks, like density integrals, help further sift through samples, such as allowing an analyst to rank samples by the density of fragmented material in a query region.

**Mixing Smoke (Figure 15).** We recreated the colliding red teapot and blue bunny smoke objects using the simulator in [Chern et al. ACM Trans. Graph., Vol. 41, No. 4, Article 156. Publication date: July 2022.]
in Houdini (128³ grid, ℏ = 0.0451, ±2 jet speeds). We built an ensemble by randomly sampling the rotational orientation of one of the objects and rasterizing smoke occupancy and color, as listed in Table 1, our WORLDPACKs resolutions match the resolution of the solver grid without need for aggressive spatial coarsening while maintaining reasonable memory footprints, and explored interesting variations and vortical structures. We could also explore the degree of smoke mixing by browsing and ranking based on the amount of purple smoke generated by a simple diffusive mixing process.

**Lava Village (Figures 3 & 16).** We sampled an ensemble of volcanic eruptions by varying parameters for a volcano’s vent shape that affect the amount of flow and angular direction (see Figure 16). As an example design goal, the animator explores where to place a small house so it is safe from a volcano eruption. Unfortunately, given the scale of this viscous FLIP-based fluid animation in Houdini, the sheer amount of simulation data is difficult to load, inspect, and query. An estimate, dumping per-timestep snapshots of lava geometry in VDB format produces roughly 1.5 GB of data for each simulation sample; more than 100GB for a 72-sample ensemble. We use this number not to compare our functionality with VDB—the authors make design decisions with a different use case in mind—but to highlight the intractability of exploring and querying multiple simulation samples using current, popular data structures. In contrast, our 72 WORLDPACKs storing geometric occupancy can fit in a total of 7.2 MB (see Table 1): comfortably in a web browser cache. Naively evaluating queries to find samples where lava does not flow through candidate house locations is expensive and can not be computed at interactive rates on this much data. Using the UMWB interface, the animator can easily find locations where the house is hit (IN query) or not hit (NOT IN query) by the lava.

**Spiral Staircase (Figure 17).** Inspired by the articulated rigid-body character falling down the stairs using ensemble refinement in [Twigg and James 2007], we consider an animation where a squishy armadillo falls all the way down a spiral staircase. It is quickly apparent that this unlikely behavior is not present in the initial ensemble: no samples reach the bottom stair. Instead, we create an ensemble in which the character is subjected to small, randomly applied impulses that hope to keep it from preemptively tumbling off the edge. The user identifies a promising outcome using the GUI, while easily discarding samples where the character has fallen off the stairs or is too close to the edge. The user then manually narrows the sampling range around the chosen outcome’s parameters, and generates a new ensemble of character motions starting from here, and repeats the process. In situ ensemble refinement remains challenging for many multi-physics animations due to high simulation costs, but parallel simulator-in-the-loop refinement is likely in subsequent works. The final result is shown in Figure 17.

**Fig. 16. Browsing Lava Flow:** (Left) We make an ensemble by varying volcano vent deformations. (Right) Combining IN and NOT IN queries allows the user to find a simulation sample where the lava flow branches in an interesting way, and uses it to create a suspenseful animation: a house in the shadow of the volcano is narrowly missed by an eruption’s lava.

**Fig. 17. Don’t Fall Off!** UMWB enables ensemble steering to identify needle-in-the-haystack scenarios. The animator’s goal is to generate an animation where a softbody character falls all the way down a spiral staircase without falling off. This unlikely scenario is not present in the first ensemble, so we instead identify a promising sample using the GUI, narrows down the sampling range to generate character motions from there, and repeat the process to steer the ensemble towards the desired result. The figure shows stills from the final outcome.

**Fig. 18. Candles:** We create an ensemble of candles lit by a flurry of particle-system sparks (Top). IN and NOT IN queries executed on the candle wicks can find samples where candles ignite or didn’t ignite. Such functionality allows the animator to explore and inspect different possible combinations of lit and unlit candles (Bottom) without needing to manually tune parameters of the particle system.
Candles (Figure 18). In this ensemble, we randomly perturb the noise, size, and trajectory of sparks to light a set of candles. We model the sparks as a turbulent particle system. At the start of the simulation, candles wicks are coated in fuel. They require a significant number of spark collisions to overcome a temperature threshold for lighting, which is difficult to control due to the noisiness of the spark particle system. We simulate fire spread in Houdini using the Pyro Source Spread solver, and we WorldPack sufficiently hot particles. Applying queries to candle wicks can search for samples where candles—or combinations of candles—did or did not kindle.

Split Milk (Figure 19). In this FLIP-based fluid ensemble, a cup of milk is spilled on a table setting by varying the cup’s angular direction, its milk fullness, and its table position. Dragging an IN query along the table, an animator can ask: “show me samples where milk has spilled furthest down the table.” Including NOT IN queries adds more specificity, such as “show me samples where milk has spilled furthest down the table, given that selected place settings stays dry” a means for interactive investigation of “what if?” scenarios.

6 CONCLUSION

We have introduced a practical approach for exploring large datasets of arbitrary physics-based animations, with the following components: a) our WorldPack representation, a unified volumetric for spatiotemporally compressed animation state optimized for data browsing, b) a query-based browsing interface that responds interactively to user input, and c) demonstrations of results on a diverse assortment of animation phenomena.

6.1 Limitations and Future Work

Most of the decisions made in our development of WorldPack were made under the assumption that we could find and exploit spatiotemporal redundancy in ensemble data. As we have shown, many simulated phenomena in computer animation (multibody dynamics, fluids, fracture, etc.) exhibit suitable redundancy: simply empty space, or repeated values, e.g., relatively constant temperature values in a spatiotemporal location, consistent normals on a piece of barely-moving scene geometry, etc. However, for simulated domains with large amounts of noise-like variations, e.g., snowstorms or sandstorms, WorldPack’s RLE-in-time compression will suffer (may even increase sizes), and more sophisticated lossless (or lossy) compression schemes may be required in such cases. Although most of our ensemble animations are short, RLE-in-time compression incurs per-octree-cell decoding overhead in the GPU ray traversal that can increase for longer or “noisier” animations.

We have used fast spacetime-AABB vs. WorldPack intersection tests for interactive queries, but more general spacetime queries are possible by encoding the query region using an WorldPack and performing more expensive WorldPack-WorldPack intersection tests.

Interactive ensemble refinement, which was possible for rigid-body simulation-in-the-loop in [Twigg and James 2007], remains a challenge for many multi-physics animations due to higher simulation costs and the lack of simulator integration in our prototype UMWB system. In addition, our WorldPack representation is immutable, and not designed to support simulation-in-the-loop ensemble refinement.

Direct GPU-accelerated visualization of the WorldPack uniform raster geometry is fast in practice, but some animation tasks may require higher visual fidelity. Alternatives could include adaptive grids, or non-raster and hybrid geometric representations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for constructive feedback; Kangrui Xue, Sofia Weytzner, and Vishnu Sarukkai for proof reading; Christopher Twigg for MWB-era discussions on fluid browsing. We thank SideFX for donating Houdini licenses which were helpful in producing examples and renders. We thank Gianluca Iaccarino, Javier Urzay, and other Stanford INSIEME group members for project feedback. This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003968.

REFERENCES


ACM Trans. Graph., Vol. 41, No. 4, Article 156. Publication date: July 2022.
There’s no use crying over it! A cup of milk is spilt on a table setting in several ways with varied start conditions. (Left) An IN query on geometric occupancy of the milk can be dragged across the length of the table to find samples where milk spills the farthest, while (Right) a combination of IN and NOT IN queries can ask more nuanced questions, such as finding samples where the milk spills furthest, subject to the constraint that certain areas remain dry.


