Visualization of the Illustris-TNG dataset with ParaView and novel TNG SpaceWalk WebApp

Bender, Nicolas and Bürg, Marc and Maul, Johannes

Abstract—This report explores the utilization of astronomical simulations, such as the IllustrisTNG Simulation, to gain insights into physical phenomena in the universe. We present some fundamental visualization techniques and a novel approach for visualizing this data using an immersive 3D web application which enables us to explore simulation data holistically. Importantly, our approach allows for real-time, interactive exploration of the data, a significant advancement over previous static visualization methods by using Octree structures and spline interpolation. Furthermore, we developed a system for handling the massive data sizes involved in these simulations by using preprocessing pipelines, allowing for the efficient transmission and rendering of data. We demonstrate the utility of our approach with various use cases, including the visualization of galaxy formations and gas outflows.

1 Introduction

One of the significant pursuits in physical research is the exploration and understanding of the universe, toward which a large fraction of effort is dedicated to astronomical simulations. Simulations, like the IllustrisTNG Simulation of astronomical dynamics, enable additional gain in knowledge compared to pure experimentally obtained observational data. One of the main resources in gathering insights from astronomical simulations are visualizations of diverse physical properties of large-scaled phenomena like galactic outflows of gas or the influence and distribution of dark matter. Visualization serves as a method to explore hidden phenomena and dynamics of simulated cosmic events. By providing an intuitive and immersive representation of numerical data, visualizations enable researchers to identify patterns, correlations, and trends that might revolutionize our understanding of cosmological circumstances. Since astronomical observations have their limitations due to distances, resolution, and instrumental constraints, visualizations can help to bridge the gap between theoretical simulations and observational data, allowing researchers to correlate their models with real-world phenomena. This synthesis of data sources allows more accurate predictions. In the last years, there have been over 600 publications based on the IllustrisTNG simulation [1]. We aim to enable a more intuitive understanding of this complex data through our visualizations and help gain new scientific insights. Big cosmological simulations yield a great amount of data, which provides the possibility to visualize structures and findings within this data. Researchers often ask specific questions and are interested in a data representation that can yield a satisfying and concrete answer, e.g. plotting certain property measurements over the time domain to visualize a certain trend. Another example might be, showing the topology of spatial structures in the case of cosmological simulations. However, fundamental three-dimensional visualizations, focusing on an easy to understand visualization approach instead of visualizing concrete scientific quantities are also helpful in many cases. Especially, when addressing non-experts, as these trivial understandable visualizations help towards sparking interest in the non-scientific community, and therefore support the goal of making new connections between experts, as well as astronomically-curious persons. Therefore, our goal is to provide insightful visualizations for both addressed audiences. Since the IllustrisTNG dataset is based on a scientific simulation, namely the AREPO Simulation, the main focus is on visualizations for the scientific community. The IllustrisTNG dataset provides various different scalar quantities like density, metallicity or masses but also higher dimensional data like velocity or magneticity. All quantities together create a complex dataset, which needs powerful visualizations to produce astronomical insights. Thus, common visualization techniques like volume rendering, stream- and pathlines are fundamental but also advanced approaches like vortex core extraction or separatrices are applied to the data. Nevertheless, no less interesting are simple visualizations like volume rendering on different quantities to discover the correlation of various properties of the universe. Furthermore, these fundamental visualizations are also easy to understand but still allow new insights into the data. In addition, the IllustrisTNG dataset allows more than just stationary exploration of the data. The included time domain adds even more complexity to the data and demands more advanced visualization techniques and hardware requirements. One of the main challenges in visualizing the simulation data is the handling of the humongous data sizes. The full size of one high-resolution snapshot of the TNG50 simulation comprises 2.7 Terabytes of data. To be able to display structures and physical properties which change over time, multiple of these snapshots have to be accessed. This comes to bear especially in terms of storage and transmission of data. For example, these amounts of data make real-time rendering of a complete simulation impossible and require sophisticated techniques to only require a certain level of detail or pre-sorted data structures. Besides the high-resolution data, the IlllustrisTNG dataset also provides partial low-resolution data. This data is easier to handle but has also downsides. The low resolution can result in bad sampling since the data is distributed over extensive areas. This low resolution is also relevant in the time domain as subsequent snapshots cover a huge time difference. This creates difficulties for the visualization of the temporal development of structures and requires accurate interpolation methods at the cost of scientific accuracy. The low resolution also bears challenges for the visualization of finer structures. These finer structures may not be represented in these low-resolution datasets and therefore can not be visualized. However, the IllustrisTNG Simulation can also trace galaxies over their lifetime. This functionality allows small areas (the area around the galaxies instead of the whole simulation) to be processed, thus neutralizing the disadvantage of low resolution. Nevertheless, the fact the IllustrisTNG dataset is a simulation recorded over time offers great potential for real-time applications. These applications visualize data over time in real-time, meaning that the user can interact with the data at an acceptable frequency. Certainly, this complicates the existing problems and creates new challenges that need to be addressed. A lot of data must be prepared in advance to simplify the data such that common hardware can handle it without losing details or real-time interaction. Also, a generic modality should be used to allow general access to the visualization. Yet, stable basic hardware is crucial to enable such visualization with such amounts of data. Hence, our main goals for this project are visualizations of the scientific insights of the IllustrisTNG Simulation data for experts implementable with various visualization techniques. In addition, we want to create a visualization for both, experts and non-experts by developing an interface that is accessible by anyone with no prior knowledge, platform-independent and with real-time exploration of the data.

Seminar Visual Computing, August 1–3, 2023.
Visual Computing Group, Heidelberg University, Germany.

[1] https://www.tng-project.org/results/

2 Related Work

2.1 IllustrisTNG

This project is mainly based on the datasets of the IllustrisTNG simulations [19]. The simulations provide large-volume time-dependent data of dark matter, gas, stars, and black holes considering gravitational, cosmological, magnetic and hydrodynamical conditions. In regards to the previous Illustris simulation [9, 24, 26], IllustrisTNG uses improved physical models and numeric refinements and extends the simulations with several features like magnetic fields and information about metals [17, 19].

First publications visualized stellar mass and gas density projections of galaxy disks, gas properties like density and temperature in galactic outflows and gas velocity fields using Line Integral Convolution (LIC). Through these visualizations, the authors were able to draw conclusions about the correlation of stellar mass and galaxy outflows as well as the emergence of complex behavior in the simulation despite simple initial conditions [18].

Apart from these insides, the IllustrisTNG project itself provides diverse types of visualization methods on its website. Here, the project offers Application Programming Interface (API) features in the form of visualization services for all simulated particle types which include gas cells, dark matter and stellar objects for subhalos and halos. Using Smoothed Particle Hydrodynamics (SPH) kernel projection, it is possible to show projections of column densities and temperature as well as line-of-sight velocity and velocity dispersion maps [19].

Besides these visualizations, the website provides an exploration feature that allows users to visualize galaxy properties of galaxies and galaxy clusters on the one hand (Infinite Galleries), and to explore large-scale structures on the other (Explorer2D and Explorer3D)[2]. Infinite Galleries generates visualizations for randomly selected galaxies at random redshifts. The user can choose to display velocity fields with LIC, V-band and H-alpha light maps, velocity maps and velocity profiles, gas in the circumgalactic medium plotted by metallicity and velocity, gas column density, temperature, pressure and x-ray luminosity, magnetic fields and properties of gaseous halos around galaxies. The web application Explorer2D offers a projection view of the TNG300 simulation in a Google Maps-style interface. It allows the user to zoom and pan in the projection and to switch between several views for properties of gas, dark matter and stars like velocity, density and temperature as well as magnetic fields and radiation. This application is realized with the JavaScript framework Leaflet[3]. The TNG Explorer3D web application provides a simulation for different TNG simulations. The visualization is a three-dimensional cube with up to 10000 halos to be loaded. Halos are displayed as wireframe spheres with a radius according to their virial radius. The user can modify different kinds of settings and aspects of visualization like the number of halos, color mapping according to a certain property, the minimum and maximum color and opacity. Another feature is "search on click" which enables the user to click on a sphere representing a halo and get information about the selected object. A second visualization method is volume rendering. This provides a representation of certain fields like gas masses displayed in a grid size from 32^3 in powers of two up to 512^3. The rendering is done by slicing the box into several subboxes, rendering front and back and casting rays that consider emission and absorption of the physical quantity they pass through. The Explorer3D application is developed in the WebGL framework Three.js[4]. Both, 2D and 3D explorers are based on pre-rendered data snippets and only provide redshift zero.

There have been over 600 publications using the IllustrisTNG dataset as of April 2023. Recent research has been on finding jellyfish galaxies in the IllustrisTNG dataset. Jellyfish are satellite galaxies with long tails of gas reaching from their stellar disk. Through a citizen-science-based survey, over 5000 galaxies could be identified visually in the IllustrisTNG datasets [30].

[2] https://www.tng-project.org/explore/

[3] https://leafletjs.com/

[4] https://threejs.org/

2.2 Star And Particle Visualization

For 3D visualization of astronomical data, there are several software available, that focus either on physical precision and realistic description or an impressive presentation and effects.

Space Engine[5] is a 3D astronomy and space simulation desktop software that allows users to explore the universe in a virtual environment. It is a proprietary closed-source project and uses real astronomical data to create a realistic representation of the universe, including stars, planets, galaxies, and other celestial objects. Users can explore the universe in real-time, travel to different locations, and create their own planetary systems. The software uses data from the Hipparcos mission containing 130,000 real astronomical objects and is supplemented by artificially created data for celestial objects, planetary surfaces and stellar textures.

The OpenSpace software [4] is an interactive open-source tool for visualizing the entire universe. It comes with a default dataset that includes information about objects in the solar system and space missions. In addition, datasets that consist of information about exoplanets and satellites as well as recent cataloging missions like Gaia can be downloaded for exploration. The software uses virtual texturing and height maps to offer detailed views of planetary surfaces. OpenSpace implements a scene graph for its internal model which can be extended by the developer community via an API with additional data.

Gaia Sky is an exploration tool for the Gaia star catalog which contains over 1.3 billion stars [22]. In this software, they offer an interactive visualization for hundreds of millions of stellar objects, planets and exoplanets, star clusters and extragalactic sources. Additionally, visualization methods are being provided like isodensity surfaces for different star types, dust and HII regions. The software aims to be used for scientific applications as well as by the general public.

2.3 Data Structure

The IllustrisTNG website provides the data in several types of structures. These structures are sorted by simulation, resolution, subbox and time step as well as affiliation to larger structures (groups and subgroups) and merger trees. The larger structures are grouped into halos and subhalos. Halos are structures comparable to galaxy clusters and consist of multiple subhalos which correspond to galaxies or accumulations of closely spaced stars, dark matter or gas. Each halo and subhalo is numbered ascending according to their mass. Over time, objects can switch their associated subhalo or subhalos merge into each other which leads to regularly changing numbering. For this reason, IllustrisTNG provides merger trees with the connection of subhalos over different time steps. Besides that, metadata provides progenitor and descendent subhalos for each time step containing the most massive subhalo. Simulations are also divided into spatially smaller subboxes containing possible objects of interest. These store objects with a higher time resolution which enables visualizations of galaxy-scale evolution.

For each simulation, data can be selected by subhalo structures and their metadata which consists of timestep, its coordinates, corresponding group, halo and tree, ID in the preceding and following time step and physical properties like mass, density and velocity. To simplify data access for larger structures, a subhalo search interface is provided.

The format of the data in the IllustrisTNG project that is provided for downloading is Hierarchical Data Format (HDF)5. This file format is designed to store and organize large amounts of numerical data and metadata. It supports heterogeneous n-dimensional datasets and allows high-performance input and output operations by providing features for fast access and optimizations for storage space. The HDF5 file format is widely used in different kinds of scientific fields. In astronomy, the most prominent examples are the LIGO gravitational waves experiment[6] and the LOFAR radio telescope data [1].

This data structure enables fast and complete download of targeted data as it is grouped by larger structures, data type and temporal development of objects. Despite this efficient storage and access, real-time applications are dependent on a more spatially and prioritized data structure for these large datasets.

[5] https://spaceengine.org/

[6] https://www.ligo.org/scientists/GRB051103/index.php

The Gaia Sky publication provides a method for visualization of over one billion stellar objects [22]. In order to be able to visualize data spatially specific and in real-time, the authors suggest an efficient data structure for spatially large datasets with many data points. The general structure is an Octree organized by the absolute magnitudes of stars from brightest to faintest. The Octree contains all stars which are sorted in spacial cube nodes where each node consists of a certain maximum number of stars and subnodes. This structure enables an efficient loading of stars depending on the coordinates of the camera and the level of detail to be loaded. Besides that, star shading is used, so far distant and faint stars can also be rendered invisible to create a more realistic visualization.

2.4 WebGL

WebGL[7] is a cross-platform JavaScript graphics library for rendering interactive 3D and 2D graphics in a web browser without the use of plugins. It is based on OpenGL Embedded Systems, a subset of the OpenGL graphics rendering API used in desktop and mobile devices. WebGL is a low-level framework that allows developers to create applications using graphics and visualizations on the web including games and large data visualizations.

There are several high-level Open Source frameworks for WebGL which allow developers to reuse commonly required functionalities in 3D visualization for web applications. It is advisable to use a framework that pre-implements common and frequently used functionalities of WebGL and makes them available to a developer. This has the advantage that many settings such as camera movements, shaders, and visualization methods are already provided. For this project, three of these frameworks can be considered: Babylon.js, ThreeJS, two general-purpose web visualization frameworks and deck.gl, a framework specialized in visualization of large datasets.

Babylon.js[8] is an open-source 3D game engine and rendering engine that is used for creating interactive 3D applications. It is written in JavaScript and is built on top of the WebGL API, which allows it to run in any modern web browser without the need for plugins or downloads. Babylon.js provides a wide range of features, including support for physics engines, lighting and shading, particle systems, and animations. It is designed to be easy to use and highly customizable by providing pre-built functions for common use cases in 3D visualizations like mesh creation and camera positioning. It abstracts the most common features so that users have little to no contact with the underlying technology WebGL. It is possible to implement your own shaders and WebGL functionality.

ThreeJS is a widely-used JavaScript library designed to simplify the process of creating 3D graphics and animations for web applications. It provides a high-level API for developers, allowing them to create complex 3D scenes without needing to implement functionalities in the lower-level graphics library WebGL. Three.js offers a comprehensive set of components required to create 3D scenes, such as geometries, materials, lights, and cameras. The library also includes utilities for loading and managing assets, such as textures, models, and animations, as well as helper functions to handle common mathematical operations, like vector and matrix calculations.

An Open-Source WebGL framework specialized on visual exploratory data analysis of large datasets is deck.gl[9]. It is mostly used for applications making use of map data as it brings many pre-built visualization layers that can be combined to create interactive and customizable visualizations.

2.5 ParaView Visualization

ParaView is an open-source application built on top of Visualization Toolkit (VTK), a widely used library for visualization in scientific computing. ParaView is designed to handle large data sets, perform parallel processing, and interactively render three-dimensional data. It supports a wide range of visualization techniques, including scalar, vector, tensor, texture, and volumetric methods, as well as advanced algorithms such as iso-surface extraction, streamline generation and volume rendering. ParaView provides a graphical user interface that enables users to construct visualization pipelines and displays output in several kinds of viewers like tabular, 2D or 3D. The visualization pipelines can contain all sorts of input data, computation steps and outputs according to the data-flow paradigm in which data is passed through the system, undergoing transformations at each stage via modules in the form of certain algorithms. This allows users to perform complex data transformations and visualizations by chaining together multiple filters. These transformations can execute plenty of operations like data clipping, slicing, or computation of contours or other derivative quantities. Each algorithm consists of input ports for data reception and output ports. A pipeline requires a producer for data ingestion, called sources. An instance of a source is a reader that imports data from files. The middle part of a pipeline contains filters for data manipulation. The last part of the pipeline, the sink, converts data into graphical primitives for rendering on a display or for storage on a disk in a separate file.

In ParaView, a pipeline module can either generate data or process input data. Data generation involves using mathematical models or reading files from disk. Data processing involves transforming input data using defined operations to produce a new output. While ParaView offers a wide range of readers, data sources, and filters, there may be cases where the available options are insufficient. To address this, ParaView allows users to add new modules through plugins. Python-based programmable filters and sources provide an easy solution for custom workflows and filters as new modules can be written as Python scripts executed by ParaView for data generation and processing. These scripts have access to Python packages like NumPy, which offer various numeric operations for data transformation.

Besides these custom filters written in Python, ParaView supports a plugin mechanism that enables the addition of new features and visualization plugins written in C++ using ParaView and VTK data processing APIs. This advanced technique offers the implementation of a wide range of high-performance algorithms but is also a challenging development as building and packaging C++ plugins that work with all distributed versions of ParaView can be complex. One example of a plugin is the Vortex Core Line filter[10] written by the Visual Computing Group of Heidelberg University. This filter computes vortex core lines of point data, for example, to get trajectories around which particles rotate. It consumes unstructured grid data and produces geometry output.

There is also an implementation of the VTK tool in JavaScript for usage in web applications[11]. The toolkit uses WebGL and supports most of the visualization algorithms from the original C++ implementation of VTK including scalar, vector, tensor, texture, and volumetric procedures.

2.6 Multidimensional Interpolation Methods

The data points of masses, momenta and energy from the dataset provided by IllustrisTNG are stored each by the summed volume quantity of three-dimensional Voronoi cells and their respective midpoint [20]. This is due to the data structure output by the simulation software Arepo. Spatial gradients are estimated by a least-squares fit to avoid errors due to discrepancies between cell centers and centers of mass.

In order to visualize the data points in three dimensions, there are several interpolation methods to obtain an accurate representation of the volume. The most accurate would be to recreate the Voronoi cells which were generated by the Arepo simulation. Since this method is very computationally expensive and visualization can also be accurately implemented by estimation using other methods, simplifications in interpolation and gradient are possible. For example, Pillepich et al. apply a simplification for gravitational calculations in which the centers of the Voronoi cells are considered to be point masses and the cell itself

[7] https://www.khronos.org/webgl/
[8] https://www.babylonjs.com/
[9] https://deck.gl/
[10] https://vcg.iwr.uni-heidelberg.de/plugins/vcg_-_vortex_core_lines
[11] https://kitware.github.io/vtk-js/

to be a sphere [20]. They also assume quantity values to be constant over the extent of a single Voronoi cell for visualization purposes.

2.6.1 Voronoi Tesselation

The Voronoi tessellation is a widely used geometric structure in computational geometry. It is capable of transforming sparse discrete particles into dense continuous functions that can be interpolated, differentiated, and integrated. In contrast to other structures like spheres, the Voronoi tessellation is space-filling and partitions the domain efficiently while minimizing the loss of structural information. Furthermore, the Voronoi cells' shape and statistics reflect the characteristic arrangement of near neighbors and the spatial distribution of particles. Algorithms used to calculate Voronoi tesselation are usually based on divide-and-conquer methods and are computationally expensive. Modern algorithms use kd-trees for spatial partitioning in order to parallelize computation [28].

2.6.2 Delaunay Triangulation

The Delaunay triangulation is a geometric construct that is the dual graph of the Voronoi tessellation. In other words, the edges of the Delaunay tetrahedra are formed by the connections between adjacent sites of the Voronoi tessellation. ParaView provides a filter that constructs a three-dimensional Delaunay triangulation from a list of input points. This filter can be used to build a topological structure from unstructured points.

2.6.3 Shepard Interpolation

Shepard interpolation is an improved method of inverse distance weighting and a widely used technique in spatial data analysis [23]. It involves estimating the value of an unknown point based on the known values at surrounding points. The estimation is performed by assigning weights to these known values, usually inversely proportional to their distance from the unknown point. These weighted values are then combined to compute the interpolated value. Shepard interpolation is particularly useful for tasks like creating smooth surfaces from scattered data points. It essentially creates a continuous representation of data across a space with accurate approximations.

3 FUNDAMENTALS

This section will cover the most important fundamental topics that are used throughout this report. Selecting the relevant topics that have to be addressed is always a hard decision. Especially choosing the depth in which they are presented. We, therefore, give the reader a short overview of the following topics, stating how we cover them. First, we focus on the IllustrisTNG project that provides the data used within our work. Here we address the goals of the project, and briefly go over the most important parts of the simulations. We will focus on providing the most important keywords and references so that the reader can understand how, why and what data is used. We list the relevant simulations, the contained fields and attributes, and the approximate size of the available data. Thereafter, we approach a few scientific visualization topics that relate once to the layout of the data, but also to the method later used throughout this report. We assume some level of familiarity with scientific visualization but also provide references to additional literature if needed. Thereafter, we shortly cover the scientific visualization application ParaView and mention its important attributes regarding our work. Finally, a few subjects connected to distributed systems and web applications are mentioned. Here we want to provide the reader with some familiarity with concepts used within the implementation of our final application.

3.1 IllustrisTNG

The data on which we operate belong to the IllustrisTNG project. This project continuously runs, evaluates, and improves on "large, cosmological magnetohydrodynamical simulations"[12]. It is based on the prior Illustlis galaxy formation model, allowing for simulations of galaxy formations, star evolution, blackhole formations, and other further astronomical relevant topics [20, 27].

The simulations utilize the moving mesh code AREPO [25], and other numerical and physical advances to achieve improved simulation accuracy, promising to solve problems, which have been present in prior simulations. Part of the core simulation techniques involve the usage of Voronoi cells, representing an unstructured mesh, following the flow of matter, while using the ARPEO library to allow for hydrodynamical simulations [20]. As such the interaction of gas cells, stars, and dark matter can be included, allowing one to study the formation and evolution of galaxies or other large-scale structures as e.g. galaxy clusters [20].

The accessible data contains a great amount of different information. There are three simulation boxes, each with a "baryonic physics" simulation run and a "dark matter only" simulation run. In our case, we focus on the baryonic physics runs. The three simulation box sizes are TNG50 (Volume 51.7 Mpc^3), TNG100 (Volume 110.7 Mpc^3), and TNG300 (Volume 302.6 Mpc^3). The smallest simulation has the best resolution regarding trace count per volume. There are also three additional lower-resolution simulation runs for each box size. Even though they might not be as interesting when trying to observe highly resolved fine-grained structures they have the advantage of smaller data sizes per time step.

Each simulation provides data for 100-time steps (called snapshots). There are also subboxes (smaller subvolumes) with a higher time resolution. In addition, the project provides subhalos (also called cutouts), which are volumes in the simulation identified by additional algorithms potentially containing interesting structures e.g. galaxies. Access to these subhalos and also how different subhalos merge over time is made accessible by the project.

The data of a snapshot step contains much information. In some cases, not all snapshots contain all simulation information but only each 5th includes key components and values such as:

1. Positions and velocities of dark matter, gas, and stars.
2. Gas properties, such as density, temperature, and metallicity.
3. Star formation rates and stellar ages.
4. Blackhole properties, including mass and accretion rates.
5. Gravitational and hydrodynamical forces.
6. Large-scale structure information, such as halo and subhalo catalogs.

Smaller snapshots focus on position, velocities, and e.g. density, omitting other properties e.g. metallicity. A full data specification is provided on their project page[13]. Snapshots of simulations, subboxes, and subhalos are provided within the HDF ([8]) container format.

3.2 Scientific Visualization

As the data of the IllustrisTNG project is in the spatial and time domain we will introduce some key components regarding visualization techniques focusing on three-dimensional space. First, we want to summarize different types of meshes and grids, focusing on the difference between a grid and a mesh. With a general differentiation between different meshes and grids, we address where the data of IllustrisTNG can be put. Here we mention a few ways of visualizing simulation attributes, e.g. position, and density in a three-dimensional representation, but also give reference to other visualization approaches, e.g. two-dimensional projection.

3.2.1 Meshes and Grids

In scientific visualization, meshes and grids are used within the representation of spatial data. They aid in regards to analysis and visualization of, in our case, three-dimensional phenomena.

With the term mesh, we refer to a polygon mesh, which is a larger structure containing vertices, edges, and faces. They are used within computer graphics as they allow for the geometric modeling of objects. They can be categorized into regular, irregular, and adaptive meshes.

A regular mesh would use equally spaced vertices and faces. Taking the center point of a regular square mesh one would receive a structured square grid [13, p. 14].

A common irregular mesh is the triangle mesh using irregular polygons, joining multiple triangles along their edges s.t. the collection forms a surface. Furthermore, there exist different kinds of meshes which consist of other basic elements (e.g. quadrilaterals).

An adaptive mesh can adapt the resolution which is beneficial if there are regions of interest that might require a finer resolution to extract relevant information. Such an approach is used within the moving mesh library ARPEO [25].

A grid represents the physical domain in a discrete form [16, p. 1]. An important requirement for different kinds of grids comes from the ability to efficiently compute quantities on those grids. The efficiency can be influenced by e.g. grid size, grid topology, or grid cell shape [16, p. 5]. One can generally divide grids into two classes, resulting in three categories. The core attribute that divides those classes is the way that grid points and the form of grid cells are organized. In a structured grid they are defined by a general rule and not through their position, meaning the connectivity between points follows a regular pattern [16, p. 10] (e.g. two-dimensional Cartesian grids). In contrast, unstructured grids have irregular nodes, while the shape of their cells can differ, allowing for higher flexibility when representing complex geometries [16, p.15]. Block-Structured grids are the third category, composed of a combination of both classes. Here a coarse unstructured grid divides the space without holes or overlaps, and each cell within this unstructured grid contains a locally structured grid [16, p.16]. This can be useful when displaying varying levels of detail.

As the IllustlisTNG simulation makes use of the AREPO moving mesh library the final data is a result based on an unstructured Voronoi tessellation. The HDF container contains a list of positions in the three-dimensional space. They represent the center of Voronoi cells. Through the usage of e.g. Delaunay triangulation ([6]) one can generate an unstructured mesh, as the polygons will have different sizes and shapes based. Therefore representing this data as a grid also requires

[12] https://www.tng-project.org/about/

[13] https://www.tng-project.org/data/docs/specifications/

an unstructured grid, or requires resampling onto a structured grid. However, this is space inefficient as only certain levels of interest require fine-grained resolution.

3.2.2 Spatial Data Structures

To visualize point data from an unstructured grid, a very simple approach could be to e.g. create meshes for each point and display them, where the structure and size might vary based on some attributes. Another approach, especially for unstructured data, where the data is not just a point in space, is using some form of spatial data structure in combination with ray casting to create a two dimensional projection of a three-dimensional structure [7, p. 1065]. Spatial structures allow for efficient querying for intersecting geometries. Examples for Spatial Data Structure are oct trees or kd-trees, both being a subcategory of Binary Space Partition (BSP) trees [7, p. 1084 ff].

An Octree recursively subdivides a certain space into eight child nodes if a given condition is met. The child nodes are also called octants. The split condition might be if the current node which bounds a certain space would contain more than N elements. All eight created child nodes are of the same size and, as the splitting continuously recursively, may also split the space again if needed. Therefore the size of an octant and also its siblings is dependent on the current depth of the tree [12, p. 14]. Adding a maximum depth as an additional constraint is also a common practice to prevent a tree from growing too deep.

Kd-trees ([3]) on the other hand partition the space recursively along a given coordinate axes. This axis-aligned splitting is an orientation constraint for the splitting hyperplane which a normal BSP tree does not require. A common way to choose the point of the hyperplane is the median of the objects inserted into the node so that the left and the right child node contain the same amount of nodes [12, p. 52]. However, there are many ways to choose these criteria. In a normal BSP tree this might be s.t. the resulting nodes are of equal spatial size [12, p. 52].

3.2.3 Volume Rendering

A spatial data structure can also be used to accelerate volume rendering, which is an essential technique in scientific visualization for exploring 3D volumetric data. Structures like Octrees and kd-trees can be used to increase rendering performance by providing efficient spatial organization and traversal.

One way is to shoot intersection rays for each pixel and sample in N depths are based on a defined distance. The intersected volume can be queried using an acceleration structure to retrieve spatially close data points. Using some interpolation method any present value can then be used to determine the color of the pixel. Based on the inclusion of intersected data points at further depth one can either visualize a volume by displaying a hard surface, or one can also create a more transparent appearance by including values at a deeper depth.

Splatting is another volume rendering approach, where a 3D volume is projected onto a 2D representation. Hansen et al. ([11, p. 158]) describe how splatting can be used to achieve an image from a list of voxel objects. They go through the objects in depth order. Here one can differentiate back-to-front and front-to-back orders.

1. Visit the voxel object and calculate the associated coordinate on the screen space.
2. Determine the point size of the voxel object
3. Apply some form of basis kernel (e.g. Gaussian kernel)

3.2.4 Projection Plots

Another way of applied visualization of the three-dimensional scattered simulation data of the IllustrisTNG data is the usage of the yt project[14]. This project provides a variety of different methods, e.g. creating a projection plot, where along a line of sight the data is integrated throughout the volume along the axis of the sight. The sum can then be normalized with the integrated spatial length to generate a two-dimensional projection[15].

[14] `https://yt-project.org`

[15] YT Docs: Visualizing - Projection Types

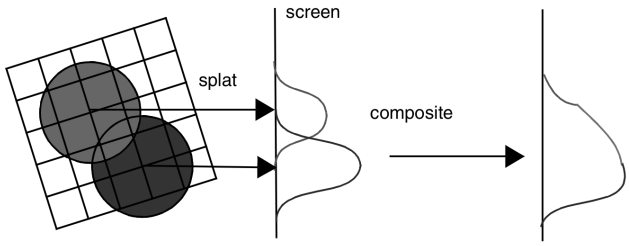


Fig. 1. Example of splatting taken from Hansen et. al ([11, p. 158]). Two voxel objects contribute in different amounts based on their depth. In addition based on other properties, they might contribute different colors.

3.3 Data Interpolation

Interpolation of data over time is a necessary ability if the timespan between data samples is high, making the change in the properties of e.g. position or velocity hard to compare to a previous point in time. A simple example might be the distance traveled by a car that was driving at around the same speed for 15 minutes. However, we only have two sample points at the beginning and the end of those 15 minutes. With interpolation, one does impose some form of assumption about the movement and position of the car during the time it was not tracked, however, it makes it easier to understand the change in distance from timestep A to timestep B instead of only showing the distance traveled between A and B, but also presenting the approximations of the points between.

For this example, simple linear interpolation might be sufficient, as we assume that the car did not experience any significant acceleration. Meaning the distance traveled changed over time linearly, as it can be expressed by:

$$\text{distance}_t = \text{distance}_a - (\text{distance}_b - \text{distance}_a) \cdot t \tag{1}$$

If the position of the car should be interpolated and we also have sampled data points for the first derivatives of the position, in this case, the velocity, one can use cubic hermite splines. Those splines allow the creation of a spline that not only connects the starting point with the endpoint but also includes the constraint that at both points a given velocity is matched. In matrix form, the position can be expressed through t between 0 and 1[16]:

$$P(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} P_0 \\ v_0 \\ P_1 \\ v_1 \end{bmatrix} \tag{2}$$

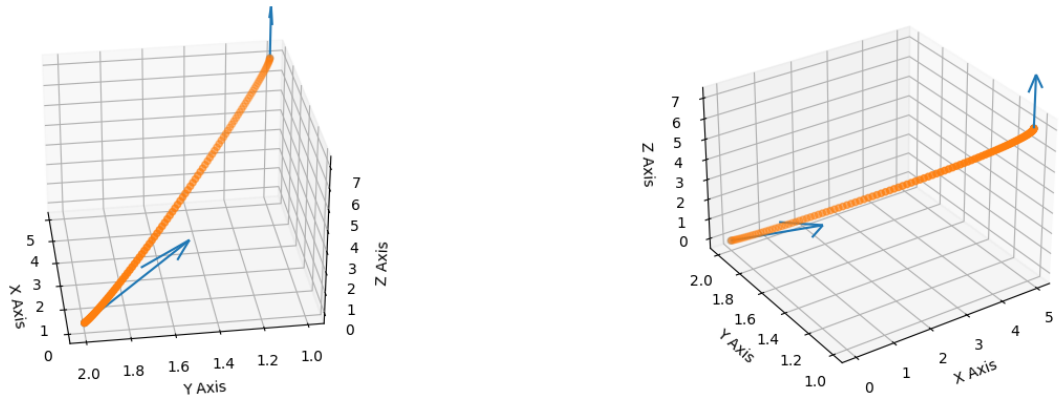
where P_0 is the starting position, v_0 is the initial velocity, P_1 is the final position and v_1 is the final velocity.

Another way of writing $P(t)$ is using the coefficients a,b,c,d ([5, p. 164, eq. 3.24]:

$$\begin{aligned} P(t) &= a \cdot t^3 + b \cdot t^2 + c \cdot t + d \\ a &= 2 \cdot P_0 - 2 \cdot P_1 + v_0 + v_1 \\ b &= -3 \cdot P_0 + 3 \cdot P_1 - 2 \cdot v_0 - v_1 \\ c &= v_0 \\ d &= v_0. \end{aligned} \tag{3}$$

In 2(a) and 2(b), one can see the same spline where the start point and endpoint, as well as their velocities, are represented by blue arrows, while the actual spline is depicted using 100 equally sampled points from starting at $t = 0$ and ending at $t = 1$.

[16] Taken from Holmér Freya: The Continuity of Splines (Snippet)



(a) (b)

Fig. 2. (a) A specific spline from a first perspective. (b) A specific spline from a second perspective.

3.4 Paraview

Applying common scientific visualization techniques can nowadays be achieved with a variety of tools and libraries. A powerful application that provides many useful built-in functionalities but also allows the extension of ParaView. It makes use of VTK, which is a library for computer graphics, scientific visualization, and image processing. It is written and designed in an object-oriented manner, trying to achieve ease of use, extendability, and user-friendliness [2, p. 3, 19ff].

Components that are of interest regarding the IllustlisTNG data are for once the Delaunay filter. It implements the Delaunay triangulation ([6]) and in combination with an appropriate data source allows the creation of an unstructured grid from the scattered input points. With this unstructured grid, one can proceed to apply other filters. Further filters that we shortly want to mention are for once are the ParticleTracer and StreamTracer.

The ParticleTracer filter implements path tracing of particles over time. It uses given vector fields e.g. a gravitational force field and a set of seed particles. The force experienced by the particles for a given time step is applied and the trajectory is calculated. The path is then displayed when going through time[17].

In contrast, the StreamTracer is static in time. It applies the forces of a given vector field onto seed particles but instead of creating a path over time, it traces the trajectory within the static vector field[18]. This is interesting when visualizing e.g. the generated magnetic field of gas particles outflowing from a galaxy. Here the static magnetic field can be used as the vector field, and the seeds can be placed around or within the outflowing gas.

3.5 3D Visualization in the Web Applications

Common scientific visualization frameworks are often domain-specific standalone programs, intended for use by scientists and domain experts. In recent years the usage of web browsers for most computing tasks done by the common user has steadily increased. For once based on the continuously improved performance, the available applications, and the ease of always using the latest version of a specific application. In comparison to native applications running on the user's computer, the web application logic is always provided by the server, this requires an active internet connection but ensures the availability of the latest improvements and features.

With WebGL, WebGL2, modern browsers provide a native rasterization API to developers of web applications. With those APIs 2D or 3D graphics can be realized. The browser itself adds support for the GLSL shader language and uses vertex and fragment shaders. Those can be set and parameterized with a platform-independent Javascript API. There is also support for WebGPU, however, the API is not stable yet, and the implementations are still unstable, making WebGL and WebGL2 more mature implementations for 2D and 3D visualization within modern web browsers. In addition other libraries, leveraging the WebGL and WebGL2 APIs, provide further convenient interfaces and out-of-the-box solutions for common problems, one faces when implementing 3D visualization. One such library is BabylonJS[19], which is also often used for game development. It provides convenient interfaces to create meshes, materials, and shaders. But also allows for easy usage of more complex helper functions and classes, e.g. a points cloud particle systems[20]. It also provides support for the creation of a user interface.

3.6 Distributed Systems

Web applications by definition are distributed systems, as the logic is executed within a browser sandbox is provided by a remote system, requesting at least an initial service that serves the application. In many cases, the served application running on the host system is also able to communicate with some sort of service proving data access and CRUD operations. Modern web applications have increased bandwidth and performance requirements, e.g. video streaming, or message exchange platforms serving many clients at once, with growing message and data sizes. Those systems demand some form of workload sharing and work distribution to allow for convenient usage. Common techniques used within the architecture of distributed systems are reverse proxies and load balancers.

3.6.1 Reverse Proxies

A reverse proxy describes a component within a distributed system, which poses a public interface towards other components within the system. It provides the user with e.g. the ability to access multiple different services under one domain rerouting requests to a specified network endpoint. It allows to hide complexity similar to an interface used within programming, but can also allow for increased security, as only the reverse proxy can be exposed to a public network, while all other services are not accessible. Therefore only services for which a

[17] https://kitware.github.io/paraview-docs/latest/python/paraview.simple.ParticleTracer.html

[18] https://kitware.github.io/paraview-docs/latest/python/paraview.simple.StreamTracer.html

[19] https://www.babylonjs.com/

[20] https://doc.babylonjs.com/features/featuresDeepDive/particles/point_cloud_system/pcs_intro

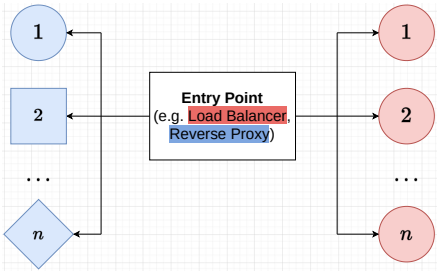


Fig. 3. Simple visualization of an entrypoint which delegates work to different endpoints. Common for a reverse proxy are diverse services, while the load balancer distributes requests among multiple instances of the same service. A combination of both is also possible.

route is specified in the reverse proxy the configuration will be open for access.

3.6.2 Load Balancing

A load balancer is somewhat similar to a reverse proxy, as it also provides a singular interface. In Figure 3 a simple generic representation for both is displayed. However, for the load balancer instead of providing routes to multiple diverse services, it groups multiple similar services to a single endpoint. The idea is to distribute similar requests to multiple service instances to increase the possible simultaneous request. The simplest load-balancing case would be a stateless request, which can be served by any arbitrary service node within a cluster of similar nodes. However, in many cases, requests are not stateless and require some form of synchronization, which is often solved by using e.g. session tokens for user identification, and database connections for stateful data access.

4 METHODS

In this section, we introduce different methods that can be used to visualize the Illustris The Next Generation (IllustrisTNG) dataset. Also, we describe the preprocessing steps in combination with the IllustrisTNG data. Finally, our WebApp is introduced with a novel approach to searching in massive, high-dimensional data.

4.1 ParaView

Although ParaView already allows many possibilities of visualization, the advantage of ParaView is the implementation of our own visualizations. subsection 3.4 describes fundamental methods provided by ParaView by default. This section focuses on the combination of built-in visualizations and new implementations. Thus, different aspects and characteristics of the IllustrisTNG dataset can be analyzed.

4.1.1 Visualizations

The IllustrisTNG dataset provides highly diverse information. A detailed description can be found in subsection 3.1. Nevertheless, the properties can be classified into two groups. First, the scalar field data with quantities like density, mass and metallicity, second the vector field data representing the velocity field or magnetic field. ParaView allows several methods to invest these different types of fields and to give a first impression of the data.

Scalar fields can be visualized by volume rendering, i.e. the data is represented as color. This coloring can be combined with the opacity, which can help to visualize important areas of the dataset by reducing the opacity of unimportant areas.

Vector fields can be visualized by glyphs which show the direction of the vector on certain points. These glyphs can also be colored e.g. based on the magnitude of the vector. Nevertheless, more advanced methods can help to analyze more complex data. Since the IllustrisTNG dataset is a complex dataset, we combine multiple built-in methods and extend them with our own implementations. This is possible because ParaView allows us to connect different methods. Thus, the visualizations can be enhanced and handle more complex data. The combination of the methods and filters is called a pipeline. This pipeline can visualize multiple characteristics for different numbers of datasets.

In Table 1 we give an overview of important filters. We also provide information on how these filters can help to analyze the data. We also explain which methods are used and which fields of data are used by the filters. Furthermore, the IllustrisTNG dataset provides every property over time, which can also be visualized by ParaView. Indeed, some filters are based on time data. Thus, we also provide this information in the table.

4.1.2 Preprocessing

The complexity and size of the IllustrisTNG dataset as well as the sensibility of ParaView about the proper data format makes it necessary to implement preprocessing routines. However, this is not only needed to handle the data itself but also to reduce the loading time and processing duration.

The IllustrisTNG dataset is structured as scattered data, ParaView does not support the raw dataset. Also, common mathematical functions get difficult e.g. interpolation and differential calculus. Since a lot of different filters are based on these functions, it is important to convert them to a proper data format, where the Delaunay triangulation comes in handy. The Delaunay triangulation converts the scattered data to unstructured data. In addition, it is a good representation for the IllustrisTNG dataset, because each scattered point represents a Voronoi cell, which is the dual representation of the Delaunay triangulation. ParaView already provides a filter, that is able to process the Delaunay triangulation on scattered data. Thus, the filter can be added in addition to the ParaView pipeline to convert the data as the first step. However, this leads to a massive processing duration, because it needs to be done for the complete timeseries, after each restart and for the complete dataset, which is likewise big. Therefore, we used the VTK framework itself to convert the data in advance and persist it, so that the unstructured grid just needs to be loaded from the disk without any calculation. We even enhanced calculation and loading performance by filtering the fields of data according to importance before preprocessing, meaning filtering the data e.g. for the density if just a scalar field is visualized.

Another preprocessing step is called resampling. The resampling step is done to convert the unstructured grid to a structured grid by performing an equidistant sampling along each dimension, i.e. converting the explicit indexing to an implicit one. This comes with the advantages of memory savings, performance optimization and easier calculations but can lead to bad resolutions depending on the sampling rate. A bad resolution is problematic because important structures vanish. Therefore, a high sampling rate should be chosen. Also, the resampled dataset can be compared with the Delaunay triangulation which should be at least similar. Again, ParaView provides a filter for resampling but this also leads to worse performance within ParaView, which is why we again used VTK to preprocess the data in advance.

Even though IllustrisTNG provides a proprietary API to download the data, we decide to implement our own control tool to get a better understanding and good control over the data. This is important because IllustrisTNG provides a lot of different datasets and the original data format of the raw data is HDF5, which chunks the data in parts after the download. Since the Delaunay triangulation needs the information of the complete dataset at once, the data needs to be combined. The handling of the IllustrisTNG dataset is implemented in the download step.

The preprocessing contains several steps and should be flexible for development purposes. Therefore, a command-line interface (CLI) is implemented, which allows control of the preprocessing steps. This includes the download step for specific simulations and timeseries, the Delaunay triangulation step and the resampling step. It is also possible to combine all steps into one pipeline such that each step runs after the other as a full preprocessing pipeline. Nevertheless, each step can be performed separately e.g. for test data.

ParaView brings the opportunity to develop our own filters as extensions by building ParaView together with these extensions. This leads to a big freedom in working with ParaView but can also lead to problems related to dependencies and building ParaView. Therefore, we set up a docker environment that automatically resolves dependencies and allows the use of the advanced ParaView build in combination with the IllustrisTNG data. Additionally, the ParaView build can be used on Windows with a Windows-Subsystem for Linux (WSL).

4.1.3 Data Management

In subsubsection 4.1.1 we introduced the different implementations of various visualization. This section describes different aspects that focus more on the IllustrisTNG dataset and its structure respectively its API.

As described in subsection 3.1 the IllustrisTNG dataset is separated into different simulations, which are simulated with different parameters and also with different resolutions. To avoid performance issues and memory shortage we use a subbox for the visualization. A subbox is a smaller area from the complete simulation. However, subboxes can also lead to other problems. On the one hand, it is not trivial to follow particles within a subbox as they can change subboxes by moving out of the bounding box over time, on the other hand, the subbox may not cover the region of interest (ROI). Thus, the subbox can be an area of the simulation that contains no interesting values. To avoid such a problem we use advanced information provided from IllustrisTNG to find a proper ROI.

IllustrisTNG provides two tools that can help to define a ROI. The first tool is the halo/subhalo finder[21], which one can find on the IllustrisTNG website. We use the filter to get a subhalo that is interesting to visualize. An interesting subhalo can be characterized by different properties like the PrimaryFlag, the stellar gas mass or the total gas mass. Here it is important to look at gas properties since we concentrate on gas properties in ParaView. The second tool to find a ROI is the merger tree tool. This is a merge tree that visualizes the lifetime of a subhalo. This is important because a subhalo can change its Identification (ID) during its lifetime because they merge or split

[21] https://www.tng-project.org/data/search/

Table 1. Important filter and their characteristics together with the inside into the IllustrisTNG dataset. Temporal types are S for stationary, i.e. not time-dependent and T for temporal, i.e. time-dependent.

Filter	Field type	Temporal type	Method	Analysis
Volume rendering	Scalar	S,T	Raycasting	Visualize data as color
Resampled volume rendering	Scalar	S,T	Raycasting	Comparison off unstructured and structured grid representation; quality of resampling
Glyph representation of vector field	Vector	S	Rendering	Visualizes the vector field for easier understanding of the structure
Dot product of two vector fields	Scalar	S,T	Raycasting	Visualizes the dependencies of two vectors on certain positions in the field
Streamline on same position	Vector	S	Runge-Kutta	Compares some vector fields by placing seeds on same position
Streamline along other streamline	Vector	S	Runge-Kutta	Visualizes the effect of one vector field on the other, e.g. the velocity on the magnetic field
Streamline on critical points	Vector	S	Runge-Kutta; [14, 21]	Visualize the field around critical points and shows it topology
Pathline	Vector	T	Runge-Kutta	Traces a vector field over time
Pathline with Streamlines	Vector	T	Runge-Kutta	Visualizes a time-dependent vector field (velocity) with a stationary vector field (magnetic)
Vortex core lines	Vector	S	[15]	Represents vortex core lines in the vector field; indicates curls
Separatrices	Vector	S	[10, 14, 21]	Represents the topology of a vector field with critical points and manifolds
Contours	Scalar, Vector	S	Iso surfaces	Visualizes important or specific areas, e.g. center of galaxy
Interpolation	Scalar, Vector	S,T	Linear interpolation	Allows to interpolate between two datasets

based on its size and neighbors. That means if we need to traverse the tree to follow a subhalo during its lifetime.

Even though these two tools are very helpful, the search is done by hand. Therefore, we implement a tool that makes use of the information of the subhalo finder and the merge tree tool to follow a subhalo programmatically and print its center of mass as a pathline. This comes with an advantage because the IllustrisTNG API also allows the download of single subhalos instead of the complete dataset. We can now use higher resolution with less memory usage, since the data focus a certain ROI. In Figure 4 the path of ten subhalos is visualized. Important to note, that one subhalo shows an outlier visible through the green dotted line. This is a problem from the IllustrisTNG merger tree, that can have problems itself, following subhalos through time.

Using a cutout of a subhalo improves performance but comes with the drawback, that the data moves over time. This circumstance makes it difficult to use filters like stream- or pathlines because they need a seed position. One possibility is to move the seed with the subhalo but this is inconvenient, difficult to calculate and can lead to bad results. Therefore, we implement a method to create an intrinsic coordinate system. Each subhalo provides different information like the coordinates of the center of mass and the velocity. We then set the center of mass always as the origin of the intrinsic coordinates system and subtract the subhalo velocity from the particle velocity, because the particle also represents the subhalo movement. This means, that the extrinsic movement is combined with the intrinsic movement. The subtraction of the subhalo velocity, i.e. the extrinsic velocity reveals the intrinsic velocity for that subhalo. Finally, an intrinsic mass-centered coordinate system is created, which allows the usage of the filters like usual.

4.2 WebApp

This section introduces our WebApp and methods that we implement to visualize the IllustrisTNG data. But it also gives an overview of the complete application and not just the visualization part, since the WebApp is developed from scratch. Nevertheless, this section focuses on the methods. The implementation can be found at subsection 5.4.

Our WebApp is separated into six main components, where four parts are looped or interactive. Also, the first five components combined build the frontend while the last component itself is the backend. The communication between different parts is done by callback functions, which work like injections, asynchronous functions or with Hypertext Transfer Protocol (HTTP) requests. The six parts are namely:

1. Initialization
2. Render loop
3. Download control and loop
4. Time control and loop
5. Graphical User Interface (GUI)
6. Backend

4.2.1 Component Description

In this section, we want to introduce the six components of the WebApp.

The first component called initialization can be seen as the startup of the web application. It setups the rendering environment, callbacks and requests initial information from the server. It also initializes the render loop, the download loop and the time loop. This is important since they need initial information like the position or the first timestep of the timeseries to execute their work.

The second component is the most important part of the visualization. The render loop executes the rendering of the scene, which includes the GUI and the particles from the IllustrisTNG dataset. Some important features are minimum and maximum color, the current interpolation time or opacity. Features like that get injected from other parts of the software via callbacks. Thus, the render loop is independent of other parts and runs without any delays. However, it also provides and controls information for the GUI and the download loop like the camera information.

The third component is called download control and this loop is mainly responsible for downloading and caching the data. It controls the communication between the front- and backend and downloads all important information based on the location. Additionally, it controls the caching for the frontend, i.e. if the camera is moving and the data is already downloaded and ready for the client, it will use the cached data instead of downloading it again.

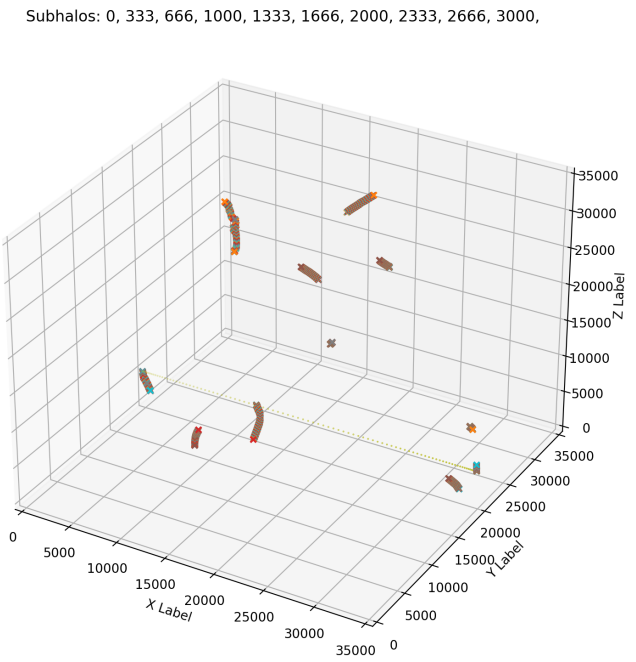


Fig. 4. Subhalo traversal over time. The crosses represent the center of mass of a subhalo for different timesteps. The dotted line represents the approximated way between two timesteps.

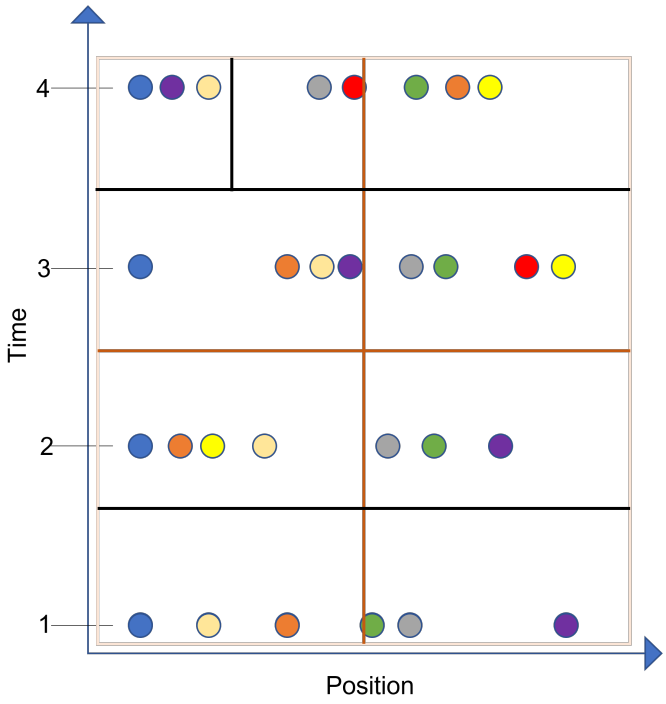


Fig. 5. 4D Octree with as 2D projection. The colored dots represent the scattered data like particles from the IllustrisTNG dataset. The color represents the particle ID. The squares represent the Octree. As darker the color as lower the level of lod. The maximum number of particles is four for each leaf.

Fourth is the time control and loop component, which controls two parts of time. On the one hand, it controls the forward-moving time, which is used for interpolation purposes. The interpolation is needed since the IllustrisTNG data provides the data as discrete timeseries. More can be found at subsubsection 5.4.1. On the other hand, it controls the current snap number, also called timestep, i.e. the time control also provides information about the current time, that is needed from the download control to download the proper dataset. This is important since the time can be changed to evolve the full potential of the IllustrisTNG simulation data.

The fifth component is the control panel for the user, called GUI. It provides different possibilities to improve the experience during the exploration of the dataset. Most callbacks and injections are performed by the GUI. This is due to the fact that nearly every changeable parameter affects the rendering. Also, it must be executed simultaneously such that the rendering loop is not interrupted by user inputs.

The last component is the backend, which is the backbone of the application. Since we want to create a web application to visualize the data, we need a backend that can be run standalone on a separate server. The backend has the responsibility to cache the data for the client in a larger scope. That means, that the backend caches the complete area, that may come to importance and responds just the data that is important for the client. Therefore, it uses a novel search structure, the *piecewise 4D Octree*. Another task of the backend is to provide information for the client, that may depend on the complete dataset, e.g. extreme values of a specific quantity like the maximum or minimum density.

4.2.2 Search Structure

The IllustrisTNG data provides a lot of data with high complexity. One dataset is the size of up to two terabytes (TBs). The smallest size is two gigabytes (GBs) but this is just for one snap, i.e. one timestep of the timeseries. A simulation can have around 100 snaps which leads to 200 GBs of memory space. Also, the data of one timestep is not ordered. That makes it challenging to interpolate or search in space. Nevertheless, we want to provide both modalities in our WebApp, locating in space and locating in time and all of that in a real-time web application. Therefore, we come up with different concepts that can tackle this problem.

We predefine some parameters that should be fulfilled or loosen the complexity:

- Scattered data is used as scattered data (no change of the grid)
- A level of detail (lod) response must be possible for position
- lod can be ordered by relevance
- Easily usable for different quantities
- Streamable
- Allow interpolation
- It must be possible to locate in space
- It must be possible to locate in time

Using these parameters make sure, that the concepts we come up with, are usable for the purpose of visualizing this complex IllustrisTNG data in real-time. The aspect of using scattered data is helpful to loosen the complexity because scattered data does not need grid information. After all, the indices can be used to address the proper information. In addition, there is no need for Delaunay triangulation or resampling steps like it is for ParaView, described in subsubsection 4.1.2.

4D Octree

One concept is the four-dimensional Octree. This way, we fit the original Octree with three dimensions to our data, because we use four dimensions from the IllustrisTNG dataset, namely the position with its x, y and z component and the time as a fourth dimension. Including the time as the fourth dimension can be done by adding the snap number as a component to the position. Every other step is similar to the original Octree. The data will be divided into cubes defining the boundaries for the next depth level of the search tree. Figure 5 represents the 4D Octree as a 2D projection of it, i.e. the position is interpreted as 1D. The projection is done because it helps to understand the concept rather than visualize the exact representation.

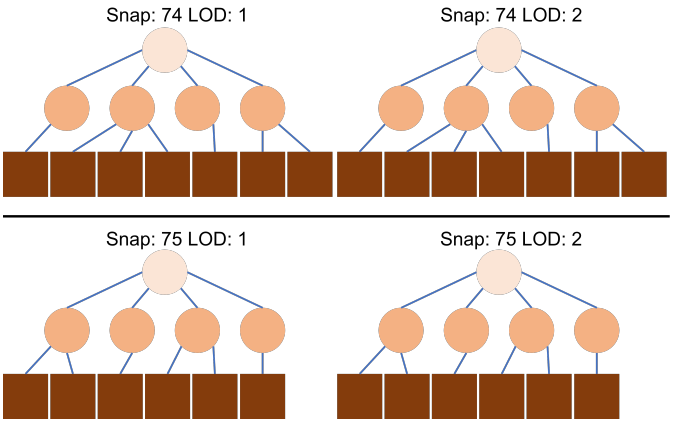


Fig. 6. 3D Octree (exemplary, could be any other search structure) for two snaps. The snaps from left to right show the lod representation, which allows streaming of different lod. The colored squares represent the node of an Octree. The lines show how the boxes are divided. The rectangles are leaf nodes and represent a set of particles. As darker the color gets as deeper the depth of the Octree.

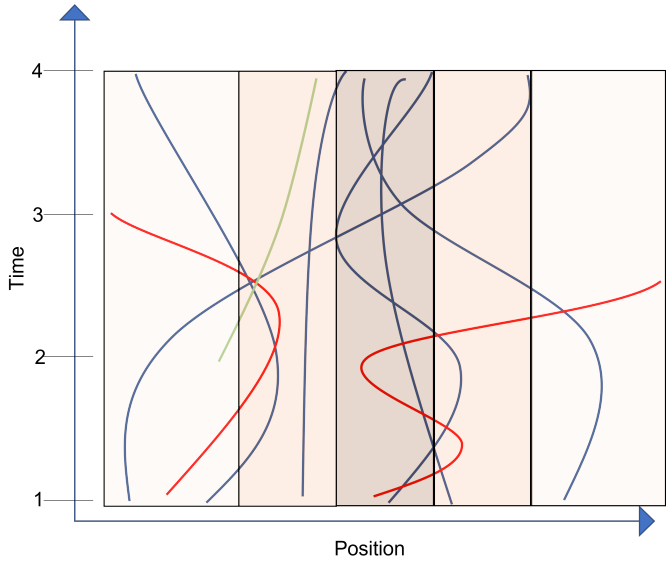


Fig. 7. Spline tree projection. Each colored line represents a particle and its movement over time. Blue lines represent normal particles; Red lines represent vanishing particles; Green lines represent appearing particles. The boxes represent different lod. As darker the color as lower the lod.

The representations show that the Octree can also be divided in time. A leaf is able to contain some particles for one snap or some particles for multiple snaps. One can say the search for space and time is combined. This approach fulfills a lot of criteria from subsubsection 4.2.2 because the search is possible in time and space, the scattered data can be used and lod with relevance is also possible to implement, but this approach also comes with some drawbacks. Firstly, the lod would create artifacts. This is caused by the fact that the lod would also consider time but should just consider the position. Secondly, all data needs to be processed at once, which is in fact not possible. Also, an Octree needs a lot of space if the depth gets too big. This problem is caused by the nature of an Octree that divides every node into eight smaller nodes. For such a big dataset the depth gets too big to store the Octree itself.

3D Snaptrees

The 3D snaptree concept is fully based on search structures that are already known like Octrees or kD-trees. Other than the section 4.2.2 the snaptrees use the 3D position data only. The fourth dimension is generated by creating a 3D snaptree for each snap. Thus, each timestep from the IllustrisTNG dataset has its own snaptree. This concept can be enhanced by creating multiple Octrees for one timestep. Each Octree represents another lod for the given timestep.

The representation shows multiple 3D snap trees. One advantage of snaptrees is the generation because it is already known from the underlying search structure, like the Octree in this case and it just uses the data of one snap, which results in smaller trees. Additionally, it reduces search times for the lod because one tree traversal is enough to specify the particles at a certain position since every tree has the same shape as the other for one timestep, even for different lod. However, this approach also comes with drawbacks. First, the time is discrete, i.e. there is no connection between timesteps, which makes it difficult to locate and interpolate in time. This is due to the fact, that the trees from two timesteps are completely different compared to each other. Thus, each particle has to be searched again when a time change is performed. Besides that, the criteria from subsubsection 4.2.2 are mostly fulfilled.

Splinetree

As mentioned in subsubsection 4.2.2, we want to tackle the time and position location problem. Even though the two concepts from section 4.2.2 and section 4.2.2 can address these location problems, both concepts focus on the position location problem. For that reason, we come up with another concept that focuses on the time location, called splinetree. Against the restrictions from subsubsection 4.2.2, this approach does not use the scattered data from the IllustrisTNG dataset. Instead, it is based on splines, which approximate the movement of the particles. In other words, the splinetree creates a search structure for splines and not for discrete scattered points. These splines come with the advantage, that they are easy to follow over time because they work like a formula with time as single input. Figure 7 shows a 2D projection of the splinetree concept.

The biggest difference is that the data is not represented as concrete points but as lines. Also the lod representation is different. Instead of squares, the division is done for the position, creating a lod for a complete timeseries. One can imagine this concept as tubes stuck in other bigger tubes and each tube represents a lod while a bigger tube also means a higher lod. This spline-based approach shifts the focus from position location to time location. If a spline is loaded the position for this spline respectively particle can be followed by just recalculating the spline equation with the new time variable, which is fast, avoids artifacts and can be advantageous against the other concept. Nevertheless, this approach has drawbacks too. First of all, the position location is difficult. Since the spline is just a formula, it is not easy to determine which particles are inside a specific area, i.e. search trees like the Octrees cannot be used for fast position location. Also, the lod definition and calculation are more complex. One extension to improve the position location could be the separation in time. Thus a search structure like the Octree could be used to search for the data, However, if the data is separated too often this splinetree concept degenerates into a 4D Octree concept from section 4.2.2, because the splinetree would also separate 4D data but instead of scattered data it would use splines instead.

4.2.3 Piecewise 4D Octree

The final concept is called piecewise 4D Octree. This concept is also implemented in our WebApp. It is a combination of the different concepts described in subsubsection 4.2.2. The underlying search structure is also an Octree because the Octree is able to tackle the position location problem in a fast way. In addition, the data is also used as time-continuous data, i.e. the tree uses splines to represent the 4D data. Nevertheless, there is a difference to the introduced 4D Octree from section 4.2.2. The piecewise 4D Octree only uses data from two timesteps. Thus, the drawback of processing the complete dataset at once can be avoided. Figure 8 shows a 2D projection of the piecewise 4D Octree.

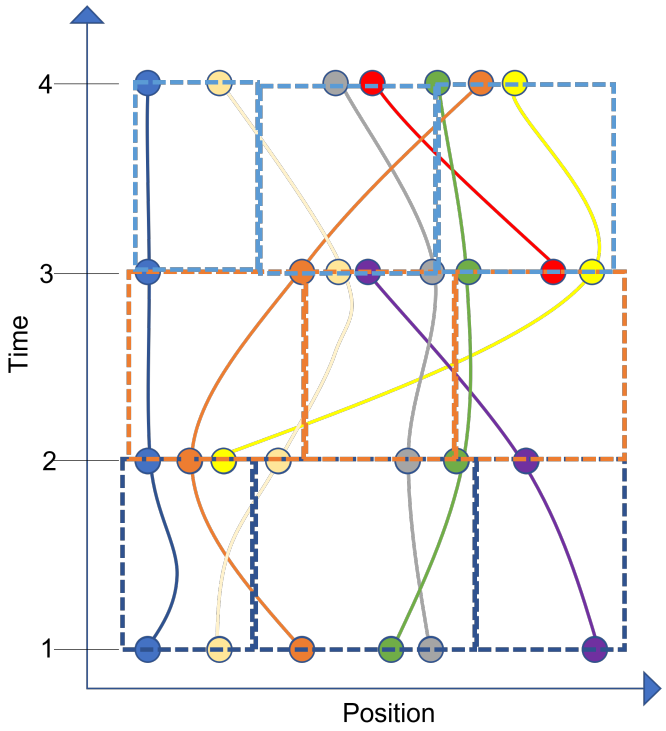


Fig. 8. 2D projection of an piecewise 4D Octree. The colored dots represent the particles of the IllustrisTNG dataset. The colored lines represent the splines of the corresponding particles. The rectangles represent the piecewise Octrees.

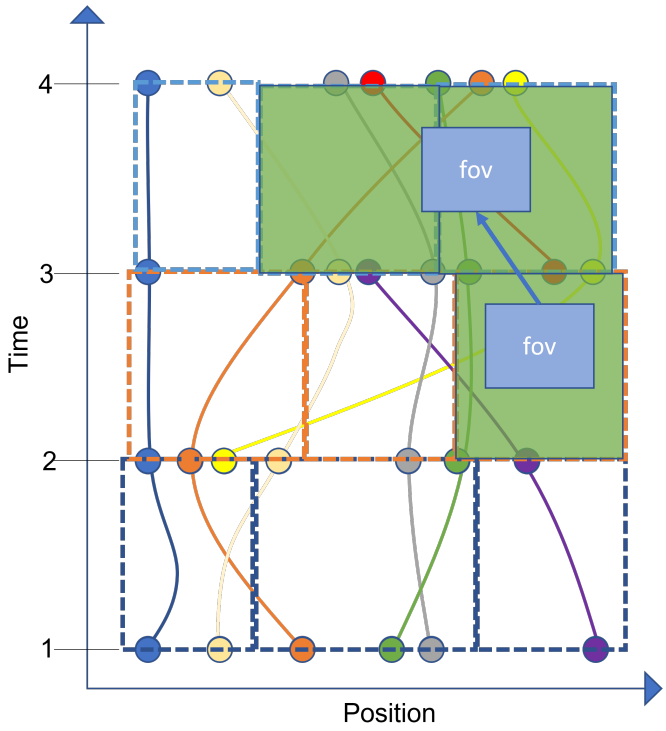


Fig. 9. 2D projection of an piecewise 4D Octree. The coloring is the same as in Figure 8 but a camera movement is visualized. The green areas mark the areas that need to be loaded because they intersect with the fov of the camera.

The piecewise 4D Octree also represents the particles as colored dots. The rectangles represent a piecewise 4D Octree for a certain timestep, i.e. if a rectangle is positioned between timestep one and two, the Octree is built for this timestep beginning at one and ending at two. Important to note is the fact, that the rectangles, cut through the spheres. This does not mean that the particles are divided or separated but it means that the particles from one timestep are used for two Octrees. For E.g. the blue dot, represented by the blue sphere, at time position two is used to cut through by the blue box and the orange box. Thus, this particle will be used for the 4D Octree of timestep one and timestep two. One can say that the vertical lines of the rectangle represent the positional division, while the horizontal division cuts through time. This concept allows combining the positional and temporal location of two snaps in one Octree because one Octree contains the positional data for two different timesteps. Thereby, the criterion of position location is fulfilled, but the time location is still discrete. Therefore, interpolation is needed, which is the reason why particles are also represented as splines. The Octree does not save the position of the scattered data but the spline information for the particles. This is similar to the splinetree concept from section 4.2.2 because splines allow the time location and enhance the interpolation possibilities of space over time. In respect of the piecewise 4D Octree, the splines are limited to represent the movement between two timesteps only. This combination of scattered data and splines, as well as other advantages like smaller Octrees caused by the fact that just two timesteps are used, tackles all the criteria from subsubsection 4.2.2. The lod handling works as usual for Octrees and the stream ability of the data is also possible. Finally, the result for a complete IllustrisTNG dataset timeseries contains $n-1$ Octrees with splines for the position and time location. All other quantities like density, mass, metallicity, etc. can be mapped to the particles via indexing. This indexing is done with a variation of the Gaia-Sky approach from [22], thus the data is ordered by relevance. However, we adapted the approach. Further description can be found in subsubsection 5.4.1. The fact, that just $n-1$ Octrees are needed is caused by the splines containing the location of the nth dataset as the end position. More can of the piecewise 4D Octree can be found in subsubsection 5.4.1.

Octree Generation

The Octree is based on particles from two timesteps. Thus, an Octree leaf contains all particles that have their movement started or ended within the box. This gets comprehensible by following the yellow particle in Figure 8. The first appearance is at timestep two. Its movement between timestep two and three shifts it from the left to the right box. This means, its movement begins at timestep two on the left box and ends at timestep three on the right box. The Octree must consider this movement because the position and time are completely free to choose. Exemplary, the assumption is done, that the camera is positioned on the right box at timestep two. The yellow particle is not visible yet. This fact can be changed by moving the time forward to timestep three. The time change moves the yellow particle into the field of view (fov) from the camera and the yellow particle must be visualized. In Figure 9 a camera movement is visualized and shows which Octree nodes need to be loaded to render the data properly. The figure also shows that time and position movement is able to handle this data structure because the viewbox moves in time and position.

However, the implementation currently just considers splines that begin or end within a box, but particles also can intersect a box for a short time. These particles are not considered. Thus, the Octree generation can be enhanced with raycasting approaches that create a mapping between splines and the boxes of the Octree they intersect. This would increase the number of particles contained by an Octree leaf but it can also increase the accuracy of the visualization.

Another aspect is the IllustrisTNG dataset itself, which comes with different problems. One problem for the Octree generation is vanishing or appearing particles. This can be understood by using *particleID* information from the IllustrisTNG dataset, which identifies a particle uniquely. These *particleIDs* can vanish or appear over time without any further information, meaning some *particleIDs* disappear from the

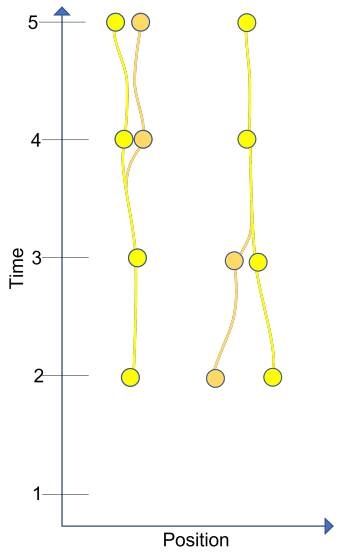


Fig. 10. Problematic behavior of particles. The left side is dividing behavior, and the right side is merging behavior.

dataset and new ones are added. Even though this behavior is intended, due to the fact that particles e.g. divide if they reach a very high density or merge if the density is very low, this behavior can lead to artifacts in our visualization. Figure 10 is a representation that shows the behavior of particles and the problem that occurs. The yellow particle will not be present in the dataset between timestep one and two. This is caused by a check we do to make sure a spline has its start and end position. Between timestep one and two we just find the end position. For timestep two and three the yellow particle is contained in both datasets, meaning a spline can be generated. This would already create an artifact since the particle appears suddenly when moving from timestep one to timestep two. For the movement between two and three anything is fine, for the division case, but between three and for particle divides. This means, the IllustrisTNG dataset now adds two new *particleIDs* and discards the old one, or it adds one additional *particleID* to the dataset. In any case, the movement cannot trace the new particles since the IllustrisTNG dataset does not provide such information, i.e. the orange spline would not be considered for the Octree between timestep three and four. This causes again an artifact when moving from timestep three to four. The visualization between four and five works properly. Summarized we can say that some artifacts can occur if another Octree is loaded, but these artifacts are caused by the changing particles from different timesteps within the IllustrisTNG dataset.

5 Implementation

In this Section, we will talk about the various components which we implemented. For each component, we give a quick reminder of what it does and why it exists with reference to the methods Section. The main focus lies on the description of each actual component, how it is structured, what it is based upon, and what for example what important libraries we utilize.

We begin with a general setup description for our work with ParaView, as we used it for the initial data preprocessing and exploration. We will also describe our implementation of the various tools, that we created regarding the preprocessing. Concerning the usage of ParaView itself we describe the most important customization to the filters that we applied and created.

The main focus of this Section will be the Web App, which allows visualization of gas cells in the form of somewhat splatter particles, in three dimensions, and over time. Giving the user the possibility to move freely in any direction. Another important part required for the functionality of this application is the effort of preprocessing the simulation data, which is also addressed. Furthermore, we will present the implementation of the distributed backend system, which tries to serve wanted data in an efficient and performance manner, Giving more insight into how the different software components interact, which libraries have been used, and how entities within the network interact.

5.1 ParaView Development Setup Effort

For our work with ParaView, we also included the implementation of the PRTL project. To allow this we needed to build ParaView from source. For this, we oriented ourselves on the documentation provided by ParaView. With the built ParaView. PRTL can then be built against the just-created ParaView instance. The documentation for this process is contained in the PRTL wiki. To create a reproducible environment we choose to create a Dockerfile which constructs a container image, containing the ParaView with the PRTL plugin. The file is fully included in the appendix (Listing 4). It requires the PRTL source code as a ZIP file in the current directory and also requires additional entrypoint scripts at the same place. All necessary data except the PRTL source code ZIP file can be found in the TNG-Starter repository[22].

We use Ubuntu 22.04 as a base image. The ParaView and PRTL documentation tend to base on Ubuntu 18.04, we, therefore, adapted the required packages accordingly. The ParaView version used is 5.10, with the timed pre-installed Python version (3.10) of Ubuntu 22.04. There are additional start scripts (`start-docker.sh`, `start-cli.sh`) in the repository, which mounts important directories into the container, e.g. the data directory used by the preprocessing code. There is also an additional share mount to easily move data from the isolated environment and the host. In the share mount we can place our preprocessing package, s.t. during each container start we are able to install the latest version of our required tools. To also allow access to GPU acceleration the script must be accordingly adapted using the appropriate docker feature dependent on the host [23]. To build a docker image one can use the Makefile in the `docker` directory, and execute the `all` target. This first generates a `Dockerfile`, based form a `Dockerfile.template`, which is then used by the docker build command to set up the environment.

5.2 Data Preprocessing for Paraview

In subsubsection 4.1.2 we talked about the way we preprocess the IllustrisTNG data. We hinted that we create a CLI interface to make this preprocessing more reproducible. The tool is written in Python using VTK and ParaView libraries and bindings to generate a processing pipeline. We also included the ability to download the data using the `requests` [24] library, following examples from the IllustrisTNG project documentation. The CLI interface is created with the `typer` [25] library to allow fast and easy addition of commands and flags.

```
# ...
# Skip if we already preprocessed the data
if not resampled_delaunay_exists(
    simulation_info
):
    # Make sure that the data is downloaded
    if not downloaded_data_exists():
        download_snapshot(
            simulation_name,
            snapshot_idx,
        )
    combine_snapshot(simulation_info)
    run_delaunay(simulation_info)
    resample(simulation_info)
# ...
```

Listing 1: Preprocessing Snippet

The downloaded data is placed in the `$HOME/Documents/data/tng` directory. We created several commands which operate on the simulation data but also extended the interface for subcommands that operate on halo and subhalos. We also implement a simple form of downloading subhalos over time, following the merge tree information by the IllustrisTNG REST API. Here we accept a starting point (snapshot ID and subhalo ID), from there on we check the API for the descendant subhalo and download it accordingly.

For simulations, halos, and subhalos we also provide the ability to apply the Delaunay filter. Herefore, we combine the wanted attributes of the downloaded HDF files, then a ParaView programmable source is programmatically created. This reads the combined HDF file, where the Delaunay filter is applied to create an unstructured grid. In Listing 1 a trimmed-down snippet is shown. This logic can then be run in parallel across multiple processes to work on multiple time steps (also called snapshots).

To ensure that the Python ParaView bindings that we use belong to our own build ParaView we use the `start-cli.sh` script with the latest wheel of the preprocessing project in the share directory. This way we can use programmable Python filters in combination with other VTK filters, e.g. `vtkResampleToImage`, or `vtkDelaunay3D`.

We also provide a CLI option which centers tracked subhalos to their center of mass, while removing the velocity dispersion from the velocity of each cell. For this, we use a combination of Numpy and the programmable Python filter[26].

5.3 ParaView

In this Section, we will cover how we used ParaView to create an initial visualization of the data, exploring different aspects, regarding e.g. the movement of large dense gas clusters over the lifetime of the universe simulation, the velocity vector fields, and how they impact the movement of gas cells, or the metallicity of the outflowing gas cells originating within a galaxy.

To visualize the density of gas cells, we used the ResampleToImage filter to create a structured uniform grid from the unstructured data generated by the Delaunay3D filter as mentioned in subsubsection 4.1.2. This is memory inefficient when done on a whole data set e.g. the whole box size of the simulation. On a spatial subset with an appropriate sample size however the required memory is manageable, and while the visualization is not exactly accurate, it is close enough to visualize larger trends. To apply other filters (e.g. StreamTracer, ParticleTracer), which focus on evaluating scalar fields, we use the unstructured grid as the base data, to achieve more accurate path and stream tracing results.

[22] `https://github.com/GimlisCode/TNG_Starter/tree/main/docker`

[23] `https://docs.docker.com/config/containers/resource_constraints/#gpu`

[24] `https://github.com/psf/requests`

[25] `https://github.com/tiangolo/typer`

[26] Centering and velocity adaption code snippet

A full list of all filters that we used within the context of this project is provided below.

To change the data layout from unstructured to structured or else we made use of the following operations.

- Delaunay3D
- CellDataToPointData
- PointDataToCellData
- ResampleToImage NO

To filter, limit, or sample the input data we use many of the already provided filters of ParaView. But with the ProgrammableFilter we make use of a versatile tool that can be used for most tasks, however, due to the execution within a Python runtime, requires more time for simple tasks, which can also be achieved faster and more convenient by e.g. the Clip filter, ergo the usage of those filters.

- Threshold
- Calculator
- Clip
- Cut
- MaskPoints
- ProgrammableFilter
- PythonCalculator

To add interpolation, visualize path and particle tracing, and add various other pre-implemented calculations to the given data we used the following list of filters.

- ParticlePathLines
- ParticleTracer
- PointVolumeInterpolator
- SPHVolumeInterpolator
- StreamTracer
- TemporalInterpolator
- ArbitrarySourceStreamTracer
- CellDerivatives

We were unsuccessful in applying the VortexCore filter provided by ParaView, therefore we choose to extract them on our own. For this, we mainly made use of the ProgrammableFilter and the PythonCalculator.

Regarding the pure visualization of interesting points, e.g. vertex points, seed points, or tubes representing a traced path we mostly relied on the below-listed filters.

- Contour
- Glyph
- TubeFilter

5.4 Web App

The Web App can be divided into a frontend and a backend. Both implementations will be described, as they have different purposes and use different technologies. A third component that is mentioned is the preprocessing for the backend. Here we extend our original preprocessing pipeline so that it can convert the TNG snapshot data into a suitable representation, which can be served by the backend, and consumed by the frontend to display a Illustrious TNG simulation.

5.4.1 Data Preprocessing

For the data preprocessing, we extended the original preprocessing tool, which we had created to work with ParaView. We enable it to download snapshots of Simulations but also download the group catalogs for each snapshot, which contain additional information about the simulations. We then save them in a new directory (`$HOME/Documents/data/tng/webapp`). The directory structure and naming convention fits the expected values used by the `illustis_python`[27]. library. This library can be used to easily load the data of a simulation. However, it requires that the data is already located and correctly named on the disk. We wrap the download logic into a CLI subcommand:

```
tng-sv-cli web download \
  --simulation-name TNG50-4 \
  --snapshot-idx 75
```

To preprocess a given snapshot n we:

1. Load snapshot n and $n+1$.
2. Extract all particles contained in n and $n+1$.
3. Calculate a gas cell diameter approximation.
4. Extract the simulation box size from the group catalog file.
5. Generate an Octree containing the relevant particles.
6. Calculate spline coefficients (a,b,c,d) for each particle.
7. Save: Octree, relevant attributes, relevant spline coefficients.

In item 2 we generate a sparse representation of the snapshot to trace which Particle IDs are within both snapshots. This is necessary, as we can only calculate the coefficients for a spline if we have the position and velocity for two snapshots. However, through this, we lose the information about the particles that are contained within one snapshot but not in the next snapshot.

For the Octree generation, we make use of the `Open3D`[28] library. They provide Python bindings to their `C++` implementation of an Octree. They do not implement a split constraint on a maximum of contained elements, but instead, only allow to specify a max depth. We dynamically calculate the maximal depth of the tree based on the box size of the simulation and a configured parameter called size per leaf, which denotes the number of elements per leaf:

$$\text{depth} = \text{ceil}\left(\log_2\left(\frac{\text{Box Size}}{\text{Size per Leaf}}\right)\right). \tag{4}$$

We also revisit each node, as the implementation of `Open3D` stores additional information about the content of child nodes, which we do not require and therefore remove. Leaving them will only increase the size of the tree. We also exchange the content of the leaf nodes s.t. they only point to a list of nodes within a two-dimensional array. This way we achieve smaller file sizes. The Octree itself is saved as a JSON object, while the two-dimensional list of nodes can be saved as two Numpy containers, once the flattened content, and secondly a scan of the length. Given the flattened Numpy array and the scan, we can reconstruct the two-dimensional list of leaf node indices.

Regarding the spline calculation in item 6 we adapt the functionality of `scipy.interpolate.CubicHermiteSpline`[29]. We remove their additional checks and class wrapping to be able to use `Numba`[30]. With `Numba` we can just-in-time compile the function, allowing us to speed up further calls. This leads to relevant time saving, as we call this function for each particle contained in both snapshots.

During the last step listed in item 7, we sort the list of particle IDs belonging to an Octree leaf node for each attribute that should be preprocessed. This way we can, later on, serve a level of detail based on the highest attribute value to the frontend, simply by sending the first N elements belonging to the leaf node. This approach is inspired by the Gaia-Sky modality [22].

5.4.2 Frontend

The frontend is implemented using BabylonJS. A graphics library that provides classes, and functions, making it convenient to create three-dimensional scenes. BabylonJS then uses WebGL, or if wanted and supported WebGL2, to render the scene.

Our project setup is a combination of BabylonJS with Typescript support, and webpack as a JavaScript bundler. We also make use of math.js for any mathematical calculations necessary in the web client. Using a bundler is common practice, especially in combination with Typescript which allows us to combine library functionality and our logic into a single JavaScript file. This makes it easier to distribute, as the browser only needs to request one file. In addition, we can also use the webpack bundler to handle the Typescript to JavaScript transpilation.

As mentioned in subsection 4.2 there are six components to the WebApp, where the first five relate to the frontend.

Interesting from an implementation point of view is the specific usage of vertex and fragment shaders within the second component

[27] `https://github.com/illustristng/illustris_python`

[28] API Docs: Octree

[29] API Docs: CubicHermiteSpline

[30] `https://numba.pydata.org/`

called the render loop. The other components are covered in the methods Section and the actual implementation is very close to the given description. As mentioned in the data preprocessing we save the information about the spatial location of data in an Octree, and also order the data elements based on some attributes. With this, we can dynamically request a specific level of detail loading from the frontend, when calling the data request endpoints of the backend. As mentioned in the methods, we send the current location for the backend to provide relevant data. The then received relevant data, is added to a Point Cloud Particle System (PCS). Where for each level of detail we add a new instance of this PCS, which is a class provided by BabylonJS, to render spatial data as single points.

However, we extend the normal approach of using this PCS, through adding information about the size of data elements, and the density, by including this information as vertex buffers in the vertex shader. The default approach would be updating the point attributes via a BabylonJS callback function, however, directly accessing such attributes within the vertex shader gives us more flexibility. Especially, as we can access the spline coefficients in the vertex shader, allowing to calculate the position, as given by the Hermite spline, allowing for interpolation in space over time. We also make use of GUI set uniform values, e.g. the scene coloring for min and max values, or other attributes that might control the scaling factor used for splatting, or used within the point size calculation. This way outside of the render loop the GUI callback sets the uniform value, which is then used in the next render step. For the creation of the GUI, we take the provided classes from BabylonJS, as they already include the most important aspects which we need to control certain parameters, that a user should be able to change, e.g. coloring, value ranges, and other scaling coefficients.

The point size calculation depends on the current scene state, as the distance between the camera position and element position is required. Furthermore, uniforms set by the user are used as coefficients in the shader e.g. the `Base Point Size`, and a scaling factor (`Scale`). Other input data as the Voronoi cell size is once again accessed through a vertex buffer and precalculated during the preprocessing step (item 3). The whole vertex shader is given in Listing 2.

$$\begin{aligned} \text{len} &= \| \text{ Position} - \text{Camera Position} \| \\ \text{Point Size} &= \frac{(\text{Base Point Size} \cdot \text{Voronoi Cell Size}) + 1}{0.02 \cdot \text{len} + 1} \cdot \text{Scale} \end{aligned} \tag{5}$$

To approximate a volumetric representation instead of only displaying single points in space we make use of a Gaussian kernel, which in combination with the gas cell diameter approximation should allow the blending of larger cells by extending their point size and overlapping their outer regions, while smoothing the hard border of the WebGL point using a kernel as a coefficient for the alpha value:

$$C_\alpha = \frac{1}{1 + (C_{\text{kernel}} \cdot \| \text{ Point Coordinate} - \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \|)^2}, \tag{6}$$

where the coefficient C_{kernel} is another user set uniform and C_α is multiplied by the alpha value determined by the density of the point concerning the current minimum and maximum density.

```
precision highp float;

attribute vec3 position;
attribute vec2 densities;
attribute float voronoi;
attribute vec3 A, B, C;
attribute vec4 color;

uniform mat4 worldViewProjection;
uniform float t;
uniform float point_size;
uniform vec3 camera_pos;
uniform float scale;

varying vec2 vdensityVary;

#define scale_coef 0.02

void main()
{

    float t_sqrt = t * t;

    // at^3 + bt^2 + ct +d
    vec3 positionNew = A*t*t_sqrt
                     + B*t_sqrt
                     + C*t + position;

    gl_Position = worldViewProjection
                  * vec4(positionNew, 1.0);
    vdensities = densities;
    float len = length(position  - camera_pos);

    gl_PointSize = (point_size * voronoi)
                   + 1. / (scale_coef * len + 1.)
                   * scale;
}
```

Listing 2: Vertex Shader

```
precision highp float;
uniform float t;
uniform vec3 min_color, max_color;
uniform float min_dens, max_dens, kernel_scale;
varying vec2 vdensities;

#define PI radians(180.0)

void main() {
    // vdens_both: density of snap t and t+1
    vec2 vdens_both = (vdensities - min_dens)
        / (max_dens - min_dens);
    vdens_both = min(vdens_both, 1.);
    float min_dens_norm = min_dens / max_dens;
    float vdensity = vdens_both[0] +
        (vdens_both[1] - vdens_both[0])
        * t;
    float one_minus_d = 1. - vdensity;
    vec3 color = min_color * one_minus_d
        + max_color * vdensity;
    float opacity = min(
        max(0., vdensity - min_dens_norm)
        / (1. - min_dens_norm),
        1.
    );

    float alphaCoef = (
        1. / (
            1.+ pow(
                kernel_scale*length(
                    gl_PointCoord - vec2(0.5, 0.5)
                ),
                2.
            )
        )
    );

    vec4 newColor = vec4(color, opacity * alphaCoef);
    gl_FragColor += min(newColor, 1.);
}
```

Listing 3: Fragment Shader

In order to visualize vector fields, we make use of an arrow representation of every data point. A combination of meshes of a cylinder for the body and a cone for the tops is used to build the arrow. The vector data to be displayed is done like follows: The coordinates are the middle point of the arrow, the direction of the vector is normalized and the arrow is rotated accordingly. It is now possible for the user to decide whether the magnitude of the vector is represented in form of the length of the arrow or in form of the color scheme of the arrow while the length of all arrows are the same. The latter alternative is preferred as the range of magnitudes comprises several powers of ten.

5.4.3 Backend

The backend architecture consists of two kinds of services. Once a load balancer service, which orchestrates multiple cache services. Secondly, a cache service, which is responsible for loading the data for a specific snapshot from disk. It returns the wanted data to the frontend based on the current spatial position, and level of detail. The load balancer keeps track of which cache services are running, and which cache service has loaded which snapshot. The frontend itself sends requests to the load balancer, which then redirects to the appropriate cache service instance, or chooses a random instance which is then responsible for loading the data from the disk. To prevent multiple cache services from loading the same snapshot at the same time the random function is based on a deterministic hash of the send request configuration:

$$
\begin{aligned}
&S := \text{Set of Hosts} \\
&i = \text{int}(\text{hash}(\text{concat}(\text{Simulation Name}, \text{Snapshot Idx}))) \\
&h = S[i \bmod \| S \|]
\end{aligned} \tag{7}
$$

.

To keep track of which cache services do exist, each cache service pings the load balancer regularly, this ensures that during the start up a cache service is registered in the orchestration of the load balancer, but also allows failure recovery, at least to some degree, if the load balancer goes down, as the cache services will be known again at some point in time.

If the load balancer encountered a new cache service, it requests the current metadata information, containing information about the loaded snapshots. If a cache service starts loading a snapshot it informs the load balancer about the current loading approach. If a cache service receives a SIGINT or SIGTERM it will gracefully send a goodbye to the load balancer, s.t. any request which previously has been served by this cache is now distributed to a different instance.

Both the load balancer and the cache service are written in Rust using the actix[31] web framework. To load the data from disk we use the ndarray[32] and ndarray-npy[33] packages, which allow us to load multidimensional data from a numpy container. In addition, we use the cxx[34] package to generate C++ bindings which are necessary for the Octree traversal. The Octree generation is done during the preprocessing step and realized with the Open3D Python package. We took the relevant C++ code for reading the Octree file, implemented the traversal in C++, and created convenient Rust bindings, which allow us to load and traverse the tree within the Rust Web App.

Each service is implemented in such a way that the stateful elements within the service is realized via the actor model[35], e.g. the data structure of the load balancer, which contains the information about which cache service loaded which snapshot, or the data cache within the cache service. The actor model uses instantiations of so-called actors which can send and receive messages to allow for the implementation of concurrent computation.

The main service provided by the backend is the actual serving of data. The most time expensive operation is the initial loading of a snapshot. The most requested operation, however, is the retrieval of data points for a snapshot, at a given point in space, with a certain level of detail. For this, the cache services use the send request information about the relevant simulation, snapshot, previous level of detail, and the current spatial position. Given this information, it executes the following logic:

1. Extract request information.
2. Load cached data.
3. Traverse Octree to find new relevant nodes.
4. Extract relevant particles based on node and current level of detail.
5. Extract data for the new relevant particles.
6. Calculate the minimum and maximum attribute values for the newly extracted data.
7. Increase the relevant levels of detail.
8. Return data for the new levels of detail.

As this is the most requested sequence of operations it must be fast. In contrast, the initial loading takes way longer (around eight seconds), hence the rather complex setup with a second load balancer, while the level of detail request is faster. The actual time consumption highly depends on the number of nodes that we cut and the number of elements per level of detail configured requested from the webclient. Here a possible optimization for clients with a bad internet connection can be to allow the user to set the number of requested elements per level of detail. Which allows faster loading at the cost of additional loading requests before all levels of detail have been reached.

[31] https://actix.rs/
[32] https://github.com/rust-ndarray/ndarray
[33] https://github.com/jturner314/ndarray-npy
[34] https://cxx.rs/
[35] https://en.wikipedia.org/wiki/Actor_model

6 RESULTS

In this section, we present our results produced by the ParaView and WebApp visualizations. Therefore, we focus our results on the visualization aspects of the IllustrisTNG dataset but also provide some performance aspects.

6.1 ParaView

In this section, we present the results from the different visualization, explain issues and give some information about different deductions. The results are based on different datasets based on time data. We also present the outcome from specific visualizations.

6.1.1 Big Volume with Low Resolution

We use different built-in filters and create various new filters to visualize and analyze different aspects of the IllustrisTNG data. First, we analyze a complete subbox from the IllustrisTNG dataset. In addition, we use a low-resolution dataset to reduce memory usage. The visualizations can be seen in Figure 11. The visualizations do provide some information but it is very difficult to deduce a correlation between different quantities. As described in subsubsection 4.1.1 densities are rendered with volume rendering techniques, represented as gray clouds. The stream- and pathlines are visualized as tubes and represent the magnetic- respectively the velocity data. Even though the data shows a density cloud, it is not possible to visualize a correlation between density and any streamlines. This is due to the fact that the ROI is chosen too big and the resolution is too low. The used data provides a region of interest with a size of $5000ckpc/h$ in each dimension and a total number of 1,171,431 particles. This problem can be seen in each picture from (a) to (d) but is well seen in a, where the tubes of the vector fields do not correlate with the density at all. In Figure (b) this also becomes clear as the pathlines move randomly within the volume. Figure (d) also does not show a correlation between the density field and the velocity field but the streamlines following the shape of a vortex. This is an indication that the vortex core line filter visualizes proper vortex cores.

Nevertheless, some information can be deduced from these visualizations. Exemplary, two density clouds can be seen in every figure and especially in figure (a), meaning the data provides two high-density areas like big galaxies. Also, we can deduce that one streamline is not enough to represent the motions of the velocity field in this big ROI. In figure (b) some separatrices and saddle points (green spheres) are shown, which indicate the topological structure of the data. However, figure (b) makes it difficult to assume a concrete structure. These problems are similar to the problems for the streamlines since the separatrices are made of streamsurfaces seeded at saddle points. The ROI is too big and the resolution too low, such that the streamsurface tracing is too rough. A similar problem is seen in figure (c), where the pathlines move randomly. This is also caused by a low resolution but in this case based on a low resolution in time. The particle movement from a pathline is too big for one certain timestep. This leads to unrelated movement between timesteps. The final Figure (d) also does not show a correlation between the density and the vortex core lines. The vortex core lines are placed at positions with low-density values. Based on the IllustrisTNG dataset, this means the particle resolution at these positions is low and the information for gas may not be relevant. However, the IllustrisTNG dataset does not just provide information about gas but also about dark matter. These vortex core lines may be an indication of black holes. Since we focused on gas data, this needs further investigation. Even though these visualizations do not give proper insight, we decided to investigate further with a higher resolution on a smaller ROI.

6.1.2 Small Volume with High Resolution

To obtain more insights from galaxies, we decide to use the possibility of the IllustrisTNG dataset to trace galaxies within a dataset to obtain the time data of a certain galaxy. I.e. it is possible to follow a smaller ROI for a high-resolution dataset. Thus, we have the same amount of particles but within a smaller area which results in a higher resolution. More concrete the data has a size of $130ckpc/h$ and a total number of 57,744 particles. To compare both datasets, we use the timestep 67 for the high-resolution dataset, because it provides fewer sampled timesteps over the simulation duration than simulations with a low resolution. The redshift of 67 is the most suitable compared to the timestep 1690 of the low-resolution dataset from subsubsection 6.1.1. In Figure 12 one galaxy is visualized. Compared to the low-resolution universe, this data gives finer structures and more concrete insights into smaller structures. The white clouds also represent the density values of the galaxy. The figures (e) to (h) visualize the same properties as the low-resolution figures (a) to (d).

The first visualization with streamlines already provides different insights. Since the resolution is higher also the vector field provides higher sampled information. Therefore, a correlation between the density and the velocity field can be seen. The velocity field follows the vortex of the galaxy until it reaches the center of the vortex. In addition, a correlation between the galaxy and the magnetic field can be seen. At first, the magnetic streamlines do not correlate with the galaxy but as further the seed points come closer to the galaxy as more the magnetic field does bend like the vortex. The length of the streamline also gets shorter when the seed points are narrow to the center of the galaxy.

In figure (f) we were not able to improve the insights. The separatrices are still difficult to generate and they do not provide information about the topological structure of the data. Nevertheless, the visualization shows some accumulations of saddle points (green spheres) at the borders of the data, which may be artifacts because the galaxy is a cutout from the original data and the borders are not well defined.

Another interesting visualization is shown in figure (g). The pathlines are seeded around the galaxy and traced for ten steps. Compared to the low-resolution figure explained in subsubsection 6.1.1, the particles follow a reasonable path. Some particles follow the vortex into the center of the galaxy, similar to the velocity field, while other particles get distracted and do not get drawn to the center of the vortex.

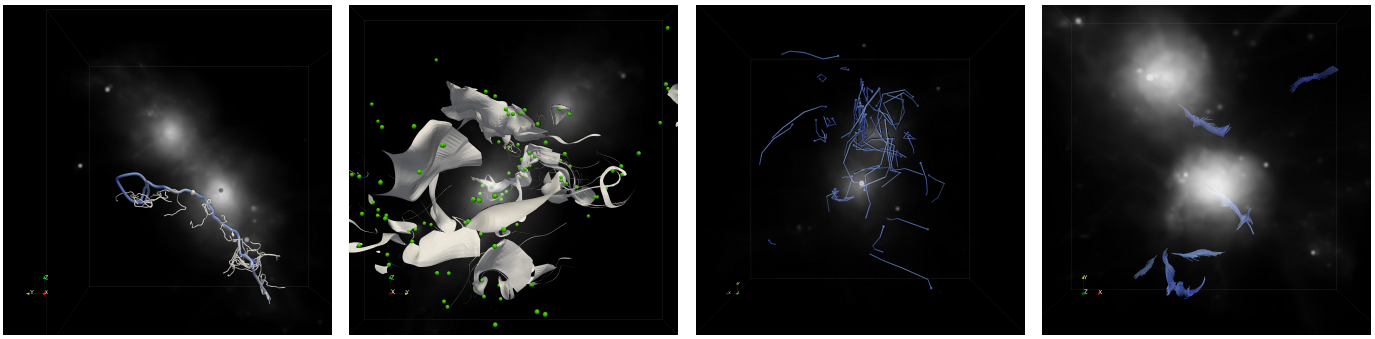
The vortex core lines are the last visualization shown in figure (h). Similar to the separatrices from figure (f), the vortex core lines do not provide reasonable information. In fact, the streamlines around the vortex cores also do not correlate to the vortex cores and do not shape as vortex like the vortices from subsubsection 6.1.1. However, in both cases, the vortex core lines do not interact or correlate with the density but should appear around the galaxy.

6.1.3 Volume Rendering

Since both approaches with advanced visualization from subsubsection 6.1.1 and subsubsection 6.1.2 do not provide the expected insights, we decide to use volume rendering with scalar quantities from the IllustrisTNG dataset. Therefore, the volume is triangulated and resampled. This is done for density and also for metallicity. Both quantities are rendered with ray casting and colored with different colormaps. This visualization is shown in Figure 13 and represents four different timesteps for the same subhalo, which is the same subhalo that is used in subsubsection 6.1.2. Again, the white clouds are used to represent the density. The yellow to red areas are metallicity. At first, the metallicity is not easy to detect in figure (a), but with advancing time more yellow clouds appear. In figure (d) the color even gets red, i.e. very old gas is released from the galaxy. Also good to see, the metal gas streams perpendicular to the galaxy disk.

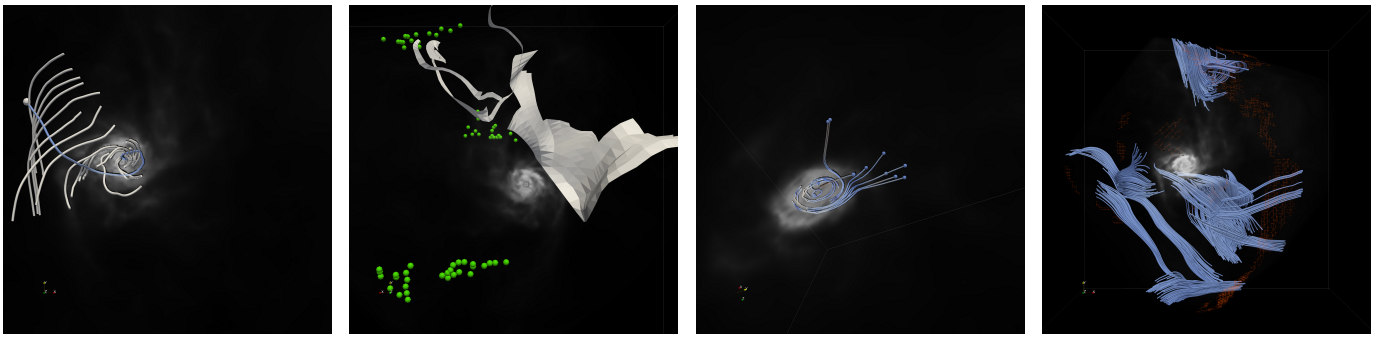
6.2 WebApp

In this section, we present our WebApp which achieved the goal to visualize the IllustrisTNG data as volume but in real-time dynamically for the corresponding ROI. Different parameters can be chosen and changed to enhance the user experience. In Figure 14 (a) the WebApp is shown. This is also the start page of the application. The WebApp starts on a default position with default values for color, maximum respectively minimum quantities and default values for the volume rendering. The WebApp allows real-time position changes, time changes or visualization changes. All settings are shown on the right side of the screen. The particles are downloaded as soon as they intersect with the ROI. The application can dynamically download batches of a specific



(a) (b) (c) (d)

Fig. 11. (a-d) Big volume, low resolution (TNG 50-4 Subbox 2): All visualizations are paused at timestep 1690 of 2330. The white clouds are volume-rendered density values based on the resampled dataset. The different figures target different visualization combinations. (a) Streamline of the magnetic field along a velocity streamline. (b) Separatrices of the velocity field in combination with saddle points represented by the green spheres. (c) Pathlines over time ongoing at timestep 1680 and also ending at 1690. (d) Vortex core lines in the velocity field with streamlines seeded to visualize the vortex.



(a) (b) (c) (d)

Fig. 12. (e-f) Small volume, high resolution (TNG 50-1 Subhalo 581059 at timestep 67): All visualizations are paused at timestep 67 from 100. The visualizations are ordered the same as (a-d) but are based on the data of one subhalo.

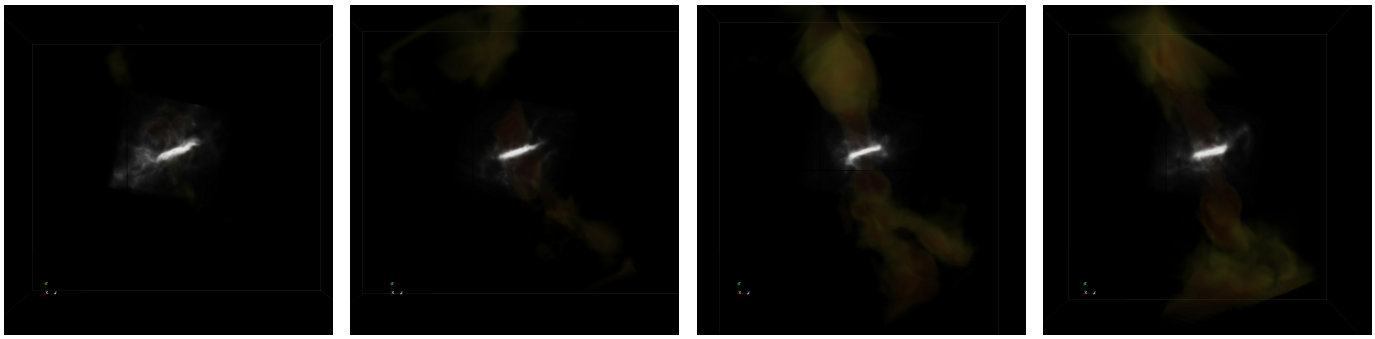
size. Thus, the download does not interrupt the real-time rendering process. This is called lod rendering. Also, it enhances the possibility of changing positions in real-time because the data is downloaded in small chunks, so the application stays interactive and position changes can be recognized and processed fast. Another tool the WebApp provides is the interpolation that moves the particles with a spline interpolation in between two timesteps as described in section 4.2.2. The interpolation just needs to be turned on in the settings on the right side of the screen. The user is also able to change the timestep manually to any preprocessed timestep, i.e. any timestep that is available on the server. Finally, the volume rendering is achieved with splatting that changes the opacity based on a specific kernel. Further readings can be found in subsubsection 3.2.3. Some settings like the initial kernel size, distance scaling and the Voronoi scale are additional settings for the splatting feature and help the user, to change the visualization in real-time. E.g. per default, the particles appear bigger if they are nearer to the camera position. This can be changed with the distance scaling parameter. It is important to note, still, all changes can be done in real-time. At the bottom of the screen, the user gets some general information like the position of the camera respectively the target, the radius, which indicates the size of the ROI and the downloading status. The frames per second (fps) indicator provides information about the real-time capability. It always starts at the highest possible but may decrease over time because the number of particles that must be rendered increases. This limitation is based on the user's hardware. Since the WebApp is a web application that can be started on every computer, it is not possible to determine which amount of particles can be loaded at maximum. Therefore, we provide an overview that explains which data is downloaded for each particle in Table 2. This data is needed from the front client part of the WebApp to render the particles. This overview

Table 2. All information downloaded for each particle and the storage information. This overview contains the data used by the front client only. No quantity unit indication means it is shown as bytes. n^1: number of loaded Octree leaves,

Information	**Memory consumption**	**Storage**
Initial settings	$<$ 2kB	Main
Loaded Octree IDs	$n^1 * 4$	Main
Loaded lod	$2 * n^1 * 4$	Main
Camera settings	$4 * 4$	Main
Particle splines	$4 * 3 * n^2 * 4$	gpu
Particle quantity	$2 * n^2 * 4$	gpu
Particle voronoi size	$n^2 * 4$	gpu
Render Settings	$<$ 1kB	gpu

makes clear, that the most data is saved on the Graphical Processing Unit (gpu) because the number of particles is the part with the highest resource consumption. Also, the main memory needs some storage for some meta information like the Octree IDs or the lod information. It also handles the data of one request, meaning it also uses the memory of one particle batch until the data is transferred to the gpu. This is no problem because the main memory surpasses the gpu memory in most cases. Therefore, the gpu memory and performance will be the limiting factor in our WebApp.

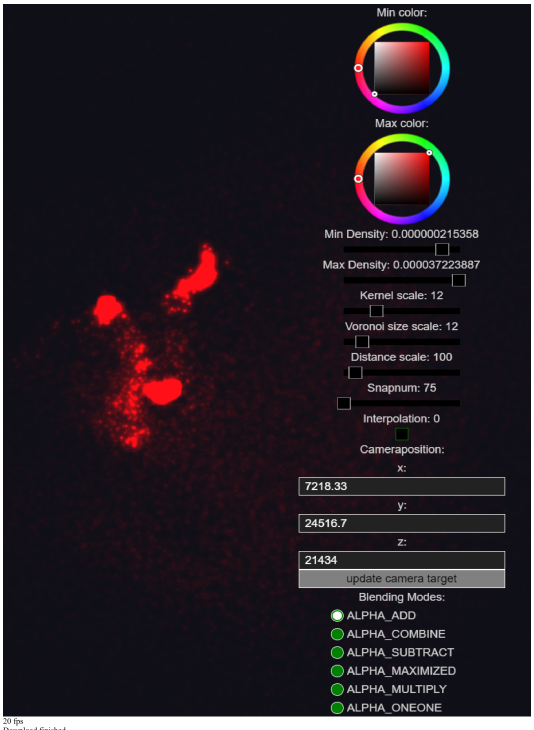
With the ability to render the data in real-time also comes the possibility to gain new insights about the data. E.g. a jellyfish galaxy can be seen in Figure 14 (b). These jellyfish galaxies are described in [29], which is a publication also based on the IllustrisTNG simulation. A



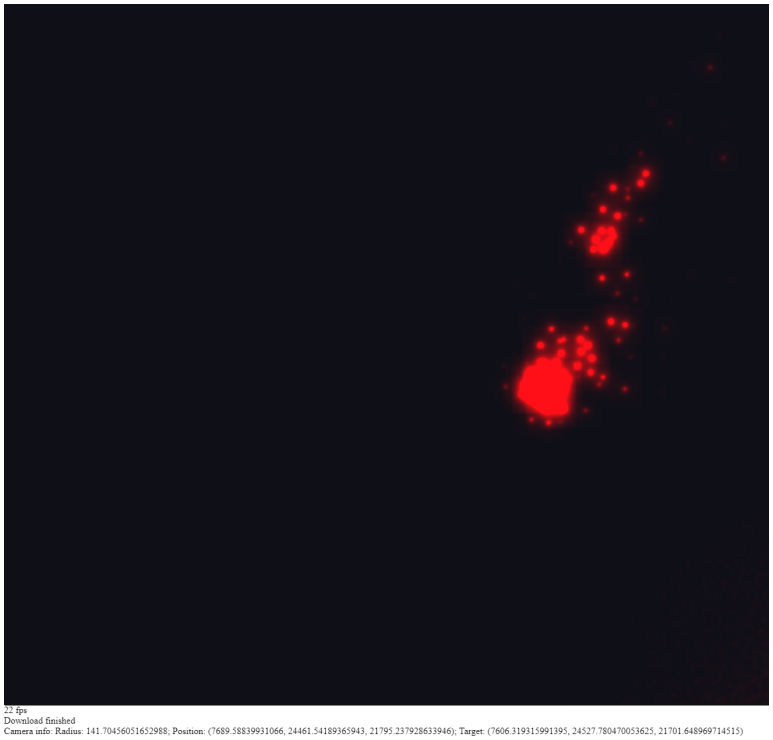
(a) (b) (c) (d)

Fig. 13. Volume rendering of the quantities density and metallicity over four timesteps. As more red the color appears, the older the gas gets. The timesteps are (a) 64, (b) 66, (c) 68 and (d) 70.

jellyfish galaxy can be described as satellites orbiting in massive groups and clusters that exhibit highly asymmetric distributions of gas and gas tails. The WebApp is able to represent such galaxies in real-time.



(a)



(b)

Fig. 14. (a) Default WebApp screen. (b) Jellyfish galaxy visualized with the WebApp.

7 Conclusion

The basis of our project lies in the IllustrisTNG project and its simulation data. The IllustrisTNG dataset provides several quantities corresponding to universes and galaxies. Related to this, the dataset covers different matters of the universe like gas, star or dark matter (PartTypes) information. All together the data is very complex. This is why we focused our work on the gas type only.

Nevertheless, the preselected data is still inherently complex, not least because of the time-dependency of the data. It provides different information represented by several scalar and vector fields. Thus, we wanted to produce visualizations that can give insights to experts but also reduce the complexity in a manner, that the interpretation is easy. ParaView was used to create several visualizations. With different filter pipelines, the complexity of the dataset was broken down by using just a small subset of the data. That led to different visualization, both stationary and time-dependent. The dataset also provides different tools to investigate different resolutions. The simulations themselves provide different resolutions but the data can also be used with subboxes in two modalities. First, it is possible to use a predefined subbox such that the data is by investing a smaller ROI. However, these subboxes do not guarantee that interesting data is included. Therefore, we implemented a tool to trace a galaxy during a lifetime. Thus, very small ROIs with high resolution can be investigated. This leads also to other visualization results as described in subsection 6.1. However, we decided to take a different direction in the project after the visualizations did not deliver the expected results despite high efforts. This can be justified by the high complexity of the data and the high demands on hardware resources.

In addition, we created a Web App during this work. That Web App called TNG SpaceWalk represents the density property of the IllustrisTNG simulations and allows a user to move through this representation in space and time. We achieved this using multiple preprocessing steps to speed up the time in which we can represent a certain time step of a simulation, and through a data structure that provides the ability to load different levels of details instead of all data at once. Using an Octree as a search structure we were able to fastly retrieve needed data based on the current point in space, viewport, and level of detail. Furthermore, we created a backend architecture that uses a specific entry point working as a load balancer s.t. different data backend services can serve a user in a more efficient manner compared to a single data service.

7.1 Discussion

Although ParaView provides a great amount of tools to visualize data, it is very sensitive to data. Also, both, the proprietary tools and the self-developed tools, tend to abort the process or they need a lot of time to finish. Preprocessing is crucial but ends up spending a lot of time on implementing the preprocessing instead of finding new information about the data.

Our preprocessing requires a good amount of computing resources for even the low-resolution simulations. Therefore, any high-resolution simulation requires a greater amount of computing resources and also an increased time effort. In addition, the compression happening through the preprocessing will still yield data sizes that are not small, meaning we require a good amount of disk space to store them, but also given the connection between the backend and the frontend client on the user side we might need a good amount of time until all level of details are loaded.

The web client also requires the user to have a somewhat powerful GPU to render scenes with a great amount of gas particles. Normal laptop hardware as used by most students will not suffice, while a normal computer also used for more demanding graphical uses, e.g. modern video games should meet the requirements given normal render settings. In addition, the current web client does not take care of limiting the total amount of data stored in the javascript sandbox for the current tab, meaning it will continuously try to load data independent of reaching hardware limits. Despite these difficulties, the choice of the framework for the frontend of the application allows dynamic extensions of further functionality. It takes most of the developer's work out of his hands by providing general functions for generating efficient renderings and predefined shaders for point clouds, meshes or lines as well as fundamental configurations for camera and user interaction handling. Also, a switch from the underlying framework WebGL to WebGPU is easily possible.

As we only focus on displaying information about the gas density we omit other interesting aspects e.g. the occurrence of black holes at certain places, which might show some correlation between certain gas properties.

Despite all that, the octree representation of data comes with a lot of advantages. Besides the fast searching capability, it also provides lod hierarchies. It inherently clusters for regions with many data samples, which allows an overview of the data distribution. We were able to combine several advantages in our novel approach.

The backend architecture using a load balancer to distribute work towards different data cache services requires the knowledge of which service stores which data. The load balancer keeps track of this information. However, with the current implementation a failing data service that does not gracefully exists would lead to a problem. Such a problem could be addressed by flushing information about this service if it is not accessible over a certain time. Determining an unavailable service could be achieved using e.g. a heartbeat functionality that monitors the health of the running services. The deterministic load balancing also requires a nongrowing size of data services to prevent requests for the same time step of a simulation, while the data for this time step is loaded. This still leaves a small margin of error in the current implementation.

7.2 Future Work

The Web App can be extended by several further improvements. The most important point is memory handling. Currently, all the data which is loaded from the server to be displayed on the client's user interface gets accumulated and never released. This leads to memory overflow at some point. To get rid of this issue, chunks of data can be flagged to outdated as soon as a critical amount of data is loaded into the user's memory respectively needs to be released. Another idea would be to limit the level of detail when the viewport exceeds a certain size.

One major point for improvement is a database for the new file format. This requires first an Octree generation for all available IllustrisTNG datasets so that every simulation can be visualized in the WebApp. These new datasets require lots of storage space and also a server capable of handling several dozens of memory to be able to load the Octree fast.

This makes an optimized preprocessing pipeline and data compression an even more important topic. A faster preprocessing speed is necessary to enable an Octree generation for multiple Terabytes of data. We see one approach for this in making use of parallelized code execution in Python for comprehensive calculations and conversions or rewriting the preprocessing code in Rust or C++. This leads to opportunities to swap code execution to multiple CPU kernels or the GPU. Another possibility for optimization lies in a better compression of the resulting data.

Another major point would be to investigate the generation of Octrees for other types of matter (PartTypes) in the simulation. At the end of this project, it is only possible to load and visualize gas-related data. Besides that, the IllustrisTNG simulations contain Dark Matter, tracers, stars and Black Holes including respective data fields. For all of these, our preprocessing can be adapted and our web framework be used to visualize these types of matter. It is imaginable to further extend the user interface in the web application to select and deselect certain PartTypes and visualize these simultaneously on one screen over time. This enables us to investigate interdependencies and correlations between different types of matter visually.

Additionally, more visualization methods can be included to allow further and deeper investigations, especially for physical and astronomical purposes beyond a layman's exploration. The idea behind this is to reimplement visualization methods that are currently available on ParaView and make them accessible on TNG SpaceWalk. The already existing framework vtk.js already provides different visualization

techniques like Surface LIC and streamline calculation which could be helpful here. Useful methods like the calculation of Vortex Core Lines would still need to be implemented or to be adapted from ParaView. These visualizations would be based either on the data that are gathered from the Octrees which would result in less accurate outcomes, but have a higher resolution temporally. Another possibility would be to send requests to the original simulation database and calculate visualizations on a dedicated server and send the results back to the user interface in the web application. This gets results with the highest possible accuracy but these would be much slower and would require another sophisticated data pipeline.

On the user's side, it is possible in the future to switch the underlying rendering framework of Babylon.js from WebGL to WebGPU. This allows us to make use of the computation kernels of the user's graphics card and accelerate the frame rate of the visualization of larger areas with more level of detail or long journeys through space. The reason why we did not use this feature yet is, that not all devices support WebGPU and we prioritize compatibility. But we hope that in the next few years, support for the WebGPU framework is given for all types of devices and browsers.

REFERENCES

[1] K. Anderson, A. Alexov, L. Baehren, J.-M. Griessmeier, M. Wise, and A. Renting. Lofar and hdf5: Toward a new radio data standard. 2010.

[2] L. Avila, U. Ayachit, B. Sebastien, J. Baumes, F. Bertel, R. Blue, D. Cole, D. DeMarle, B. Geveci, W. Hoffman, B. King, k. Krishnan, C. Law, W. Schroeder, K. Martin, L. Avila, and C. Law. *The VTK user's guide*, vol. 11. Kitware, 2021.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] A. Bock, E. Axelsson, K. Bladin, J. Costa, G. Payne, M. Territo, J. Kilby, M. Kuznetsova, C. Emmart, and A. Ynnerman. OpenSpace: An open-source astrovisualization framework. *The Journal of Open Source Software*, 2(15):281, July 2017. doi: 10.21105/joss.00281

[5] R. L. Burden, J. D. Faires, and A. M. Burden. *Numerical analysis*. Cengage learning, 2015.

[6] B. Delaunay et al. Sur la sphere vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7(793-800):1–2, 1934.

[7] J. D. Foley, F. D. Van, A. Van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*, vol. 3. Addison-Wesley Professional, 2014.

[8] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, pp. 36–47, 2011.

[9] S. Genel, M. Vogelsberger, V. Springel, D. Sijacki, D. Nelson, G. Snyder, V. Rodriguez-Gomez, P. Torrey, and L. Hernquist. Introducing the illustris project: the evolution of galaxy populations across cosmic time. 09 2015.

[10] T. Günther and I. B. Rojo. Introduction to vector field topology. *Mathematics and Visualization*, 2021.

[11] C. D. Hansen and C. R. Johnson. *Visualization handbook*. Elsevier, 2011.

[12] V. Havran. *Heuristic ray shooting algorithms*. PhD thesis, Ph. d. thesis, Department of Computer Science and Engineering, Faculty of . . . , 2000.

[13] B. Jähne, H. Haussecker, and P. Geissler. *Handbook of computer vision and applications*, vol. 2. Citeseer, 1999.

[14] J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, 285:69–94, 1995. doi: 10.1017/S0022112095000462

[15] Y. Levy, D. Degani, and A. Seginer. Graphical visualization of vortical flows by means of helicity. *AIAA Journal*, 28(8):1347–1352, 1990. doi: 10.2514/3.25224

[16] V. D. Liseikin. *Grid generation methods*, vol. 2. Springer, 2010.

[17] D. Nelson, A. Pillepich, S. Genel, M. Vogelsberger, V. Springel, P. Torrey, V. Rodriguez-Gomez, D. Sijacki, G. Snyder, B. Griffen, F. Marinacci, L. Blecha, L. Sales, D. Xu, and L. Hernquist. The illustris simulation: Public data release. *Astronomy and Computing*, 13:12–37, 2015. doi: 10.1016/j .ascom.2015.09.003

[18] D. Nelson, A. Pillepich, V. Springel, R. Pakmor, R. Weinberger, S. Genel, P. Torrey, M. Vogelsberger, F. Marinacci, and L. Hernquist. First results from the TNG50 simulation: galactic outflows driven by supernovae and black hole feedback. *Monthly Notices of the Royal Astronomical Society*, 490(3):3234–3261, 08 2019.

[19] D. Nelson, V. Springel, A. Pillepich, V. Rodriguez-Gomez, P. Torrey, S. Genel, M. Vogelsberger, R. Pakmor, F. Marinacci, R. Weinberger, L. Kelley, M. Lovell, B. Diemer, and L. Hernquist. The IllustrisTNG simulations: public data release. *Computational Astrophysics and Cosmology*, 6(1), May 2019. doi: 10.1186/s40668-019-0028-x

[20] A. Pillepich, V. Springel, D. Nelson, S. Genel, J. Naiman, R. Pakmor, L. Hernquist, P. Torrey, M. Vogelsberger, R. Weinberger, and F. Marinacci. Simulating galaxy formation with the IllustrisTNG model. *Monthly Notices of the Royal Astronomical Society*, 473(3):4077–4106, oct 2017. doi: 10.1093/mnras/stx2656

[21] F. Sadlo. Scientific visualization lecture, 2021.

[22] A. Sagristà, S. Jordan, T. Müller, and F. Sadlo. Gaia sky: Navigating the gaia catalog. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):1070–1079, 2019. doi: 10.1109/TVCG.2018.2864508

[23] D. Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*. ACM Press, 1968. doi: 10.1145/800186.810616

[24] D. Sijacki, M. Vogelsberger, S. Genel, V. Springel, P. Torrey, G. F. Snyder, D. Nelson, and L. Hernquist. The Illustris simulation: the evolving population of black holes across cosmic time. *MNRAS*, 452(1):575–596, Sept. 2015. doi: 10.1093/mnras/stv1340

[25] V. Springel. E pur si muove: Galilean-invariant cosmological hydrodynamical simulations on a moving mesh. *Monthly Notices of the Royal Astronomical Society*, 401(2):791–851, 2010.

[26] M. Vogelsberger, S. Genel, V. Springel, P. Torrey, D. Sijacki, D. Xu, G. Snyder, D. Nelson, and L. Hernquist. Introducing the Illustris Project: simulating the coevolution of dark and visible matter in the Universe. *MNRAS*, 444(2):1518–1547, Oct. 2014. doi: 10.1093/mnras/stu1536

[27] R. Weinberger, V. Springel, L. Hernquist, A. Pillepich, F. Marinacci, R. Pakmor, D. Nelson, S. Genel, M. Vogelsberger, J. Naiman, and P. Torrey. Simulating galaxy formation with black hole driven thermal and kinetic feedback. *Monthly Notices of the Royal Astronomical Society*, 465(3):3291–3308, nov 2016. doi: 10.1093/mnras/stw2944

[28] G. Wu, H. Tian, G. Lu, and W. Wang. Parvoro++: A scalable parallel algorithm for constructing 3d voronoi tessellations based on kd-tree decomposition. *Parallel Computing*, 115:102995, 2023. doi: 10.1016/j.parco.2023.102995

[29] K. Yun, A. Pillepich, E. Zinger, D. Nelson, M. Donnari, G. Joshi, V. Rodriguez-Gomez, S. Genel, R. Weinberger, M. Vogelsberger, and L. Hernquist. Jellyfish galaxies with the IllustrisTNG simulations – i. gas-stripping phenomena in the full cosmological context. *Monthly Notices of the Royal Astronomical Society*, 483(1):1042–1066, nov 2018. doi: 10.1093/mnras/sty3156

[30] E. Zinger, G. Joshi, A. Pillepich, E. Rohr, and D. Nelson. Jellyfish galaxies with the illustristng simulations – citizen-science results towards large distances, low-mass hosts, and high redshifts, 2023.

Acronyms

API Application Programming Interface

BSP Binary Space Partition

CLI command-line interface

fov field of view

fps frames per second

GB gigabyte

gpu Graphical Processing Unit

GUI Graphical User Interface

HDF Hierarchical Data Format

HTTP Hypertext Transfer Protocol

ID Identification

IllustrisTNG Illustris The Next Generation

kB kilobytes

LIC Line Integral Convolution

lod level of detail

PCS Point Cloud Particle System

ROI region of interest

SPH Smoothed Particle Hydrodynamics

TB terabyte

VTK Visualization Toolkit

WSL Windows-Subsystem for Linux

A Code Snippets

```
FROM ubuntu:22.04

ENV DEBIAN_FRONTEND=noninteractive
ENV PARAVIEW_BUILD_EDITION=CATALYST

RUN useradd -m USERNAME

RUN apt-get update -y && apt-get upgrade -y
RUN apt-get install -y git cmake build-essential \
    libgl1-mesa-dev libxt-dev qtbase5-dev qt5-qmake \
    libqt5x11extras5-dev libqt5help5 qttools5-dev \
    qtxmlpatterns5-dev-tools libqt5svg5-dev \
    python3-dev python3-numpy libopenmpi-dev \
    libtbb2-dev ninja-build ffmpeg zip unzip \
    libboost-all-dev

USER USERNAME
WORKDIR /home/USERNAME

# Get source code
ADD --chown=USERNAME:USERNAME ./prtl-master.zip ./
RUN unzip prtl-master.zip

RUN git clone --recursive https://gitlab.kitware.com/paraview/paraview.git
RUN cd paraview && git checkout v5.10.0 && git submodule update --init --recursive

# Build Paraview
RUN mkdir build

WORKDIR /home/USERNAME/build

RUN cmake ../paraview -GNinja -DPARAVIEW_USE_PYTHON=ON \
    -DPARAVIEW_USE_MPI=ON -DVTK_SMP_IMPLEMENTATION_TYPE=TBB \
    -DCMAKE_BUILD_TYPE=Debug
RUN ninja -j22

# Build prtl
WORKDIR /home/USERNAME
RUN mkdir prtl-build
WORKDIR /home/USERNAME/prtl-build
RUN cmake -GNinja -DParaView_DIR=../build/lib/cmake/paraview-5.10 \
    -DPRTL_ENABLE_CUDA=OFF ../prtl-master
RUN ninja -j22

USER root
RUN apt-get install -y nano vim python-is-python3 python3-pip sudo
RUN usermod -aG sudo USERNAME
RUN echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers

USER USERNAME
RUN pip install --upgrade pip
RUN pip install scipy tqdm numpy h5py

ADD --chown=USERNAME:USERNAME ./entrypoint.sh /entrypoint.sh
ADD --chown=USERNAME:USERNAME ./entry.py /entry.py
ENTRYPOINT ["/entrypoint.sh"]
CMD bash
```

Listing 4: Dockerfile template to compile ParaView with PRTL