OpenSpace: A System for Astrographics

Alexander Bock, Emil Axelsson, Jonathas Costa, Gene Payne, Micah Acinapura,
Vivian Trakinski, Carter Emmart, Cláudio Silva, Charles Hansen, Anders Ynnerman

Fig. 1. Using OpenSpace to depict astronomical phenomenæ at different spatial scales. Visualization of the Apollo service module in front of Earth (left), volumetric rendering of a solar wind density simulation in the heliosphere (center), exploring galaxy clusters using data from the Sloan Digital Sky Survey including the missing data due to the shadow of the Milky Way (right).

Abstract—Human knowledge about the cosmos is rapidly increasing as instruments and simulations are generating new data supporting the formation of theory and understanding of the vastness and complexity of the universe. OpenSpace is a software system that takes on the mission of providing an integrated view of all these sources of data and supports interactive exploration of the known universe from the millimeter scale showing instruments on spacecrafts to billions of light years when visualizing the early universe. The ambition is to support research in astronomy and space exploration, science communication at museums and in planetariums as well as bringing exploratory astrographics to the class room. There is a multitude of challenges that need to be met in reaching this goal such as the data variety, multiple spatio-temporal scales, collaboration capabilities, etc. Furthermore, the system has to be flexible and modular to enable rapid prototyping and inclusion of new research results or space mission data and thereby shorten the time from discovery to dissemination. To support the different use cases the system has to be hardware agnostic and support a range of platforms and interaction paradigms. In this paper we describe how OpenSpace meets these challenges in an open source effort that is paving the path for the next generation of interactive astrographics.

Index Terms—Astrographics, astronomy, astrophysics, system.

1 Introduction

The night sky has long piqued the imagination of humans and inspired our collective quest for knowledge. From ancient mariners to Galileo to modern day astronomers and astrophysicists, increases in knowledge have led to greater awareness and newly formed questions about the universe around us and our place in it. Visualization has played an enabling role in the development of our perception of the universe by providing tools for understanding and communication of these findings to our peers. This forms the foundation for the field of *Astrographics* which encompasses the use of visualization for exploration as well as explanation in the space and astronomical domain.

Despite many efforts to develop software for astrographics, the field is now facing serious challenges in terms of size and complexity of data from observations and simulations, and standard visualization tools are not providing the required capabilities. Additionally, the state-of-the-art in visualization is progressing very rapidly and it is a challenge for these tools to capitalize on the development of visualization using new methodology and hardware solutions. We are also seeing a rapid confluence of visualization methods traditionally used in exploratory visualization and those used to explain scientific findings [33], which opens up new possibilities in science communication as well as enriching the scientific exploration of data.

In this paper we describe the open source effort, *OpenSpace*, with the mission to support the use of astrographics in science communication, space exploration, and astronomical research. One of its foundational requirements is the need for multi-platform support and provisioning of tailored tools for usage scenarios ranging from individual desktop systems to planetariums, as well as presentations and collaborative and distributed visualization across multiple sites world-wide.

The underlying idea behind OpenSpace is to develop and maintain a versatile system that lends itself to rapid development cycles for implementation of new functionality and features, thus shortening the time between scientific discovery and dissemination. While at the same time be robust enough for deployment in research and public use in museums, science centers, planetariums, and class rooms.

Previous efforts have resulted in extensions to a different planetarium software, Uniview [24], but due to the closed nature of proprietary software these extensions were not incorporated into the released version. This leads to the loss of rapid prototyping efforts, which an open source platform helps to mitigate. The open source approach also enables a curated community effort managed by a core team of developers.

- *Alexander Bock and Anders Ynnerman are with Linköping University and the University of Utah. E-mail: {alexander.bock | anders.ynnerman}@liu.se.*
- *Emil Axelsson is with Linköping University. E-mail: emil.axelsson@liu.se.*
- *Jonathas Costa and Cláudio Silva are with New York University. E-mail: {jccosta | csilva}@nyu.edu.*
- *Gene Payne and Charles Hansen are with the University of Utah. E-mail: gpayne@sci.utah.edu and hansen@cs.utah.edu*
- *Micah Acinapura, Vivian Trakinski, and Carter Emmart are with the American Museum of Natural History. E-mail: {macinapura | vivian | carter}@amnh.org.*

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

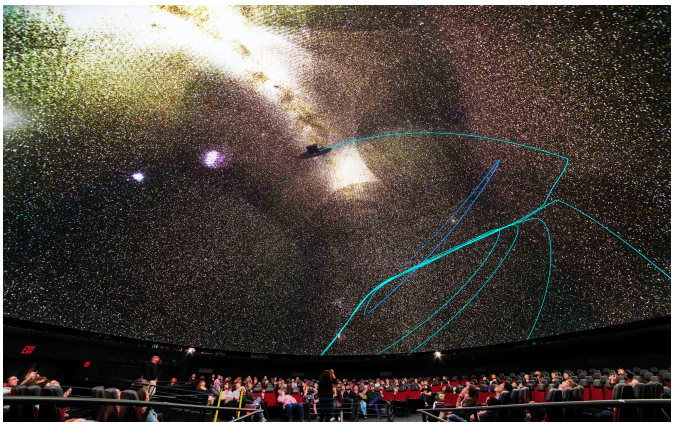

Fig. 2. An interactive visualization session with astrophysics domain experts presenting the 1.3 billion stars of the Gaia DR2 data set during the 2018 NYC Gaia Sprint at the Hayden planetarium in New York.

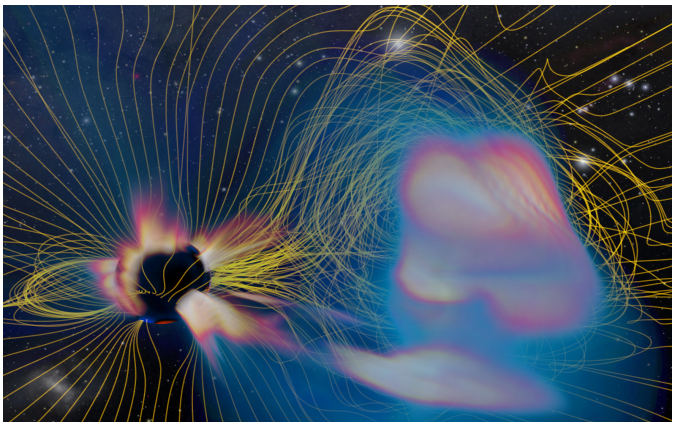

Fig. 3. Using OpenSpace as a tool for astronomy research, here in the use case of visualizing a time-varying coronal mass ejection simulation, combining a volumetric rendering and fieldlines.

1.1 Astrographics use cases

In this section we provide three use cases representing different contexts in which OpenSpace serves as an enabling software system in the field of Astrographics. The cases have been used throughout the development cycles of OpenSpace to define challenges and test solutions in an iterative participatory design process involving the core developer team, student projects, and domain experts in the astronomical sciences as well as in science communication and education.

1.1.1 Astrographics as a tool in astronomy research

Data generated through observation and simulation is rapidly growing in size and complexity. A recent example is the use of OpenSpace to visualize data from the Gaia instrument [10] containing more than 1 billion stars in the Milky way, each with complex attribute data, (Figure 2). Another example is simulation data ranging from MHD simulations of space weather in the heliosphere [8] (Figure 1(b) and Figure 3) to simulations of galaxy formation at cosmological spatial and temporal scales [9]. The ambition to support interactive analysis of this data in a consistent representation of the whole universe, providing accurate context for the data, poses challenges in terms of handling of spatio-temporal data in complex scenes using mixed mode rendering of geometry, volumes, and point-based data. Additionally, tailored tools for interaction and navigation must also be available to support the workflow of the domain expert.

1.1.2 Astrographics in planetariums, exhibits, and classrooms

One of the core uses of OpenSpace is to support novel visualizations in planetariums and pave the way for new science communication paradigms in these immersive environments. A typical use case is a facilitated exploration of a curated set of astronomical objects and phenomenæ (Figure 1(b) and Figure 2). This puts demands on rendering quality and speed, as well as use of intuitive visual metaphors to represent these complex and abstract concepts. User-friendly navigation tools that allow for smooth transitions in space and time and selection of objects are needed to support a planetarium presentation. Additionally, the same content can also engage the public in museum settings using display-walls built from multiple monitors and multi-user/multi-touch table interfaces [6]. Here the self explanatory, non-linear, and interactive story telling embedded in the installations becomes a central component in providing a rewarding visitor experience. It is also recognized that space exploration and astronomy are an integral part of school curricula and have served not only as a topic in itself, but also as a carrier of interest for mathematics and physics. The use of OpenSpace, data availability, and technical maturity provides new pedagogical opportunities for students to interactively explore these topics. The use of advanced software solutions in a class room setting do, however, require significant adaptation and integration of learning goals and interaction design principles tailored for educational situations.

1.1.3 Mission visualization for communication and planning

Visualization is not only used for data visualization in astrographics, but also to visualize the space missions that capture this data. OpenSpace offers the possibility to provide the concurrency and details of spacecraft operations and instrumentation to the audience, as well as to tell stories of current and past space exploration to the general public. For example a recreation of the Apollo missions based on recently captured lunar surface data and historical data records of the missions make it possible to recreate the historical landings on the moon (Figure 1(a)). The same methodology can also serve the purpose of contextualizing spacecraft planning for teams of researchers and developers or indeed enable in-situ visualization of data from instruments on spacecraft (Figure 4) creating new opportunities in comprehensive and universal views of observed and simulated data. The use of space mission data calls for support of high precision location of spacecrafts and celestial bodies in space and time and interfaces to a range of data sources provided by space agencies and research institutes.

2 Related Work

There are several planetarium vendors that supply software packages for the visualization of the solar system and the universe. One of these is a product from Sciss; Uniview [24]. Uniview provides an interface to multiple data sources, such as the American Museum of Natural History's Digital Universe catalog. While Uniview grew from a research project, it has since been spun off into a company where

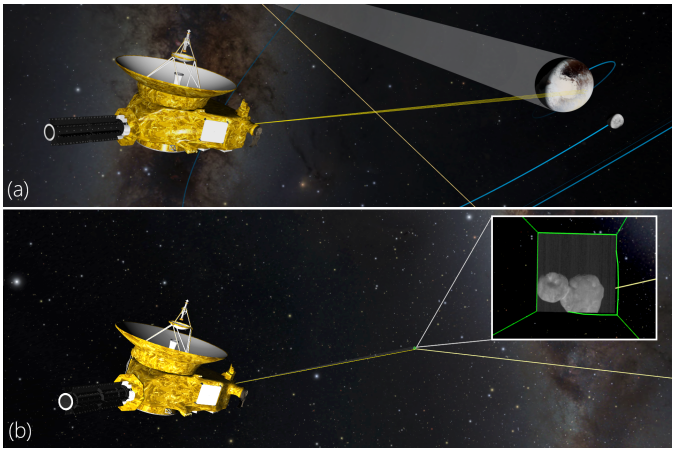

Fig. 4. The New Horizons spacecraft fly-by of Pluto in 2015 at around 20000 km (a) and its fly-by of MU69 (Ultima Thule) at about 6500 km in 2019. Pluto is about 2400 km in diameter, Ultima Thule is 31 km.

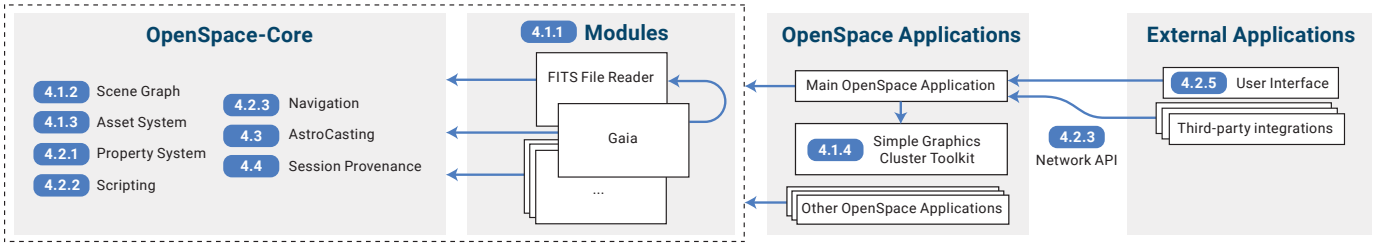

Fig. 5. The system architecture is divided into four layers. Arrows denote dependencies and rounded rectangles indicate sections in which the respective subsystems are described in detail. OpenSpace consists of a core component, multiple, each optional, modules which are combined to build an executable application.

source code access was subsequently curtailed, leading to a closed software package without the ability for users to add completely new functionality. Since one of the goals of OpenSpace is to enable an extension of the capabilities of the system, a closed software package does not meet this goal. Another planetarium vendor supplied package from Evens and Sutherland is Digistar [15]. Like Uniview, it is a closed software package without the ability for user extension. Skyskan delivers and supports the planetarium software package DigitalSky 2 [29] that runs on Skyskan's Definiti theater systems. It, too, is a closed system where extensions are built and incorporated by the vendor, though limited capabilities for user-generated scripting is possible.

WorldWide Telescope [30] was designed to be a rich virtual observatory for the visualization and sharing of data from major observatories, telescopes, and astronomy institutions. World Wide Telescope was developed by Microsoft Research and is now an open source project managed by the American Astronomical Society. It provides capability for visualizing the solar system, stars, guided tours of the universe, but focussed on presenting the data from Earth's viewpoint.

Celestia [12] is an interactive visualization system showing the solar system and objects in the universe. It provides virtual texturing for high-resolution imagery of planetary surfaces and accurate positioning of celestial bodies. Celestia is an open source system released under GPL and was started in 2001. Pre-compiled programs exist for Linux, MacOS, and Windows and are released in irregular cycles (average time between releases 16.5 months since May 2002). The last release occurred in 2011, but development activity recently restarted.

Stellarium [31] is an open source software in which the user can view the sky in 3D from the surface of any planet and the software provides a star catalog, deep space objects, time control, and the ability to add artificial satellites. Like Gaia Sky and World Wide Telescope, it is primarily a system for astronomy and allows users to add content.

NASA Eyes [22] is a suite of applications developed by the Visualization Technology Applications and Development Team at NASA's Jet Propulsion Laboratory and California Institute of Technology. The various applications, which are based on WebGL, provide tools for exploring Earth, the Solar System, Exoplanets, and others. While providing rich content from NASA missions, unlike OpenSpace the Eyes executables are non-extensible applications.

Gaia Sky [28] is an open source system designed to visualize and analyze the stars in the Milky Way gathered from the ESA Gaia mission. Currently, it can visualize 1.3 billion stars through the use of a novel, magnitude-space LOD octree data structure. Gaia Sky provides capabilities for visualizing the universe but lacks some of the features of OpenSpace such as globe browsing [4], the ability to read and use NASA SPICE kernels for space missions other than Gaia, and the ability to incorporate other data sources, such as volumetric data sets.

Another software that originated as a scientific collaboration is the *CROSSDRIVE* project, which focuses on the exploration of surface features on Mars [18]. The software provides GIS tools to visualize elevation data, ability to define landing ellipses, and inspect volumetric data sets. Additionally, it has the ability to place surface rover models into the virtual terrain. The project is aimed at creating a distributed virtual environment for these rovers using telepresence technology that supports a collaborative decision-making process for scientists and engineers [17, 27]. It produced results for three use cases, namely data analysis for Martian atmospheres, characterization of rover landing sites, and rover target selection during its real-time operations. CROSSDRIVE provides high-resolution surface terrain rendering similar to OpenSpace's globe browsing, but uses the HEALPix map projection format instead [19]. However, it lacks many of the other features of OpenSpace, such as rendering capabilities for the greater universe, and support for other planetary bodies in the solar system. Unlike OpenSpace, CROSSDRIVE is not open source.

3 Challenges

There are several aspects and challenges that need to be addressed when designing and implementing a system for astrographics. Here, we give a summary of the key challenges addressed in the development of the OpenSpace system.

Spatial and Temporal Scales Spatial scales range from millimeters on models of spacecrafts to billions of light years reaching out to the end of the universe. As the scales exceed the normal floating point representation, special care needs to be taken to avoid imprecision in rendering and navigation. The time scales of interest are also ranging from seconds in the changes of the solar wind to billions of years in the cosmological perspective. This calls for navigation and interaction aides to identify points of interest for both space and time and enable multiple representations of the same object/phenomenon.

Variety of data sources The ability to dynamically load and access different types of data from varying sources is critical to depict past, current, and anticipated events in astrographics rendering. Data may range from raw observation data, such as imagery from satellites or space probes of planetary surfaces and star fields, to derived information originating from observations, such as positions and trajectories of celestial bodies including planets, stars, galaxies, and exoplanets. Moreover, the ability to incorporate simulation data is key for astrographics, for example simulations of solar wind or the simulation of galaxy formation. Trajectory and event data from space missions are necessary for mission planning and presentation.

Collaborative Experiences The ability to share sessions between planetariums allows events, such New Horizons Pluto fly-by (Figure 4), to be shared between planetarium presenters, mission scientists, and the public. At times, it is important for scientists to remotely work together whether they are in different laboratories or in the same building, and an astrographics systems needs to account for this requirement.

Flexibility and Robustness Given the wide scope of applications and data sources, a system for astrographics must be easily and dynamically configurable and extensible. This requires effective data management, and support for incorporation of new rendering modules and means of interaction. As OpenSpace is deployed in production at public venues, the demands on reliability and robustness are at the same level as any commercially available software product.

4 The OpenSpace System

To deal with the myriad of challenges that arise in astrographics, we designed the OpenSpace architecture to be flexible and extensible at multiple levels (Section 4.1). *Developers* are able to write new *Modules* (Section 4.1.1) that can implement entirely new features, interaction

methods, or rendering techniques. *Builders* are able to use the *Asset* system (Section 4.1.3) to combine existing modules created by *Developers* to create new scenes and visualize new data sets using existing methods. OpenSpace addresses the challenge of supporting a variety of display devices from desktop displays to power walls to planetarium domes (Section 4.1.4). Lastly, *Users* can utilize existing scenes created by *Builders* to explore data and use the scripting (Section 4.2.2) and the user interfaces (Section 4.2.5) to manipulate available scenes and either explore the available data self-driven or use it as a presentation tool.

To enable collaborations, OpenSpace provides the ability of *Astrocasting* to link together shared sessions for collaborative experiences (Section 4.3). OpenSpace provides for session provenance and easy playback through the recording of scripting, control data, assets, and camera paths (Section 4.4).

4.1 Architecture

There are four layers to the architecture of OpenSpace which were influenced by addressing the challenge of flexibility, see Figure 5.

OpenSpace-Core The `OpenSpace-Core` software package contains components to manage the rendering, to define multiple orthogonal APIs to support extensibility, and determines the overall application flow. Included in this is support for command line parsing, OpenGL rendering, texture loading, font rendering, defining abstract base classes for the module interface, networking APIs, as well as, support for scripting components (Section 4.2.2), windowing configuration for display (Section 4.1.4), and infrastructure to load scenes at runtime (Section 4.1.3). Additionally, the core provides the scene graph handling (Section 4.1.2), handles user interaction (Section 4.2), and controls the flow of the rendering component. The `OpenSpace-Core` is written in C++ and is a monolithic subcomponent of the OpenSpace system.

Modules Modules in OpenSpace are self-contained packages that provide specialized instances of C++ classes used for rendering and data management, provide scripts to influence the rendering of objects that are defined in a module, and respond to user input. The objects defined in a module are used to initialize scene graph nodes when loading a scene. Modules are described in greater detail in Section 4.1.1.

OpenSpace applications The main OpenSpace application uses the `OpenSpace-Core`, the enabled modules, and the selected windowing framework and creates the actual executable file that is used by builders and users. Moreover, other utility applications use the same libraries to facilitate collaborative experiences (Section 4.3) and to provide command line tools for developers.

External applications Through the definition of a network API, it is possible for external applications to communicate and control an OpenSpace session. For example, the OpenSpace user interface based on the Chromium Embedded Framework is using this API to communicate with the main application (Section 4.2.5). The network API allows customized third-party user interfaces to interact with OpenSpace.

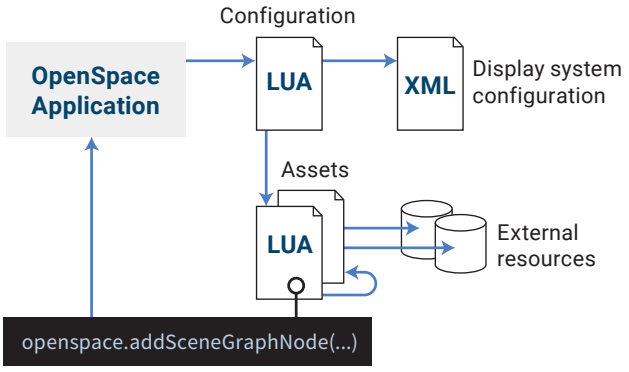

Fig. 6. Loading of display configuration and the asset graph involves a scripting language (Lua), which enables a simple, yet powerful, runtime configuration of the scene graph.

OpenSpace is combining these four building blocks into a modular architecture with customization that manifests on three levels. First, the software system has modularity at compilation time, where entirely new modules can be added by developers and are enabled and disabled independently. The selection of enabled modules results in an application that has a particular set of functionality, for instance determining whether there is support for volume rendering or not. The selection of enabled modules results in a single compiled program. Figure 6 shows an overview of the remaining two customization methods available at the start of an OpenSpace session. The scene composition is defined through an asset graph specified by *Asset* files that determine the content of the underlying scene graph at runtime and determine which of the defined renderable objects is used in a particular instance of OpenSpace. Assets contain descriptions of scene graph nodes and also define scripts to be executed at initialization time to define key bindings, which data to load, determine visual properties, and other runtime attributes (Section 4.1.3). The display system configuration specifies the number of image generators in a rendering cluster, the number of windows per device, and the type of rendering for each window. This flexibility enables the same application to drive a single window on a desktop computer to a multi-pipeline rendering cluster displaying in a planetarium without the need for specialization.

The main scripting language in OpenSpace is Lua, which is a simple, yet powerful language [21] that is often used as a drop-in scripting language for games [20]. We chose Lua as the scripting language, as opposed to Python, since it only possesses a reduced number of data structures and a simple syntax, which makes it possible for non-programmers to edit scripts that perform efficiently at the same time. OpenSpace provides a large number of Lua functions in a helper library that makes common operations easy for non-programmers.

4.1.1 Modules

Additional functionality is added to OpenSpace through the use of an extensible module system. The usage of distinct, self-contained modules enables a separation of concerns that increases the resilience when distributing modules between developers and to the public. It also allows developers to extend the functionality of OpenSpace while requiring only minimal knowledge of the rest of the system, thus decreasing the effort required to incorporate new functionality. The trade-off is that modules can only perform the set of OpenSpace actions that are defined through the API in the `OpenSpace-Core`.

Modules provide this customization by adding additional functionality through an API defined by the `OpenSpace-Core`. Modules can only have a compile-time dependency on the `OpenSpace-Core` and other explicitly named modules. In addition, modules are allowed to have dependencies on third-party libraries, as those dependencies would only impact the module itself and other modules that depend on it. One such third-party library is GDAL, the Geospatial Data Abstraction Library [32], that is part of the pipeline used to render planetary surfaces and the globe browsing component [4]. Apart from the explicit dependency between modules, all modules have to be optional and a requirement of OpenSpace is the ability to successfully compile without any modules enabled even though the resulting application would not be able to perform any visualization tasks.

Modules can combine the definition of new transformation classes, renderable classes, third-party dependencies, and utilization of multiple callbacks throughout the frame rendering exposed by the `OpenSpace-Core` to perform a large number of possible actions. Modules can provide new Lua functions (Section 4.2.2) that are available for users of OpenSpace. Once registered, these Lua functions are available to be called either though the built-in command line console or through integrated or external graphical user interfaces.

An example is the `Gaia` module that contains a `Renderable` subclass that renders an octree-based subdivision of a large star catalog, such as the Gaia dataset, contains Lua scripts to preprocess the data, and helper classes that are not used by `OpenSpace-Core`, but simplify the code and can be used by future dependent modules. Another example is the `Webbrowser` module that includes the Chromium Embedded Framework used to render OpenSpace's user interface (Section 4.2.5).

4.1.2 Scene Graph

While the specification of the scene graph follows commonly used patterns, OpenSpace utilizes the approach presented by Axelsson *et al.* to handle the large scale differences that occur in the universe by changing the traversal scheme of the scene graph nodes during the rendering step [3]. This method is centered around a dynamic traversal strategy that treats the currently selected focus node (Section 4.2.4) as the origin of the transformation traversal. This method achieves high numerical precision for objects that are close to the focus node regardless of their location relative to the global scene graph origin.

OpenSpace maintains a single scene graph that represents the parent/child relationship of all loaded objects. As with a traditional scene graph, each node has a transformation component that consists of `Translation`, `Rotation`, and `Scaling` subcomponents, each of which are optional. In addition, each scene graph node has an optional `Renderable` subcomponent that implements an interface for rendering to screen. Each of the `Translation`, `Rotation`, `Scaling`, and `Renderable` subcomponents is part of the developer API and is specified in concrete modules using a factory pattern. Scene graph nodes that have a `Renderable` subcomponent are called *Render Nodes*. If they do not possess the subcomponent they are `Internal Nodes` that can be used to group transformations for child objects, for example satellites orbiting a co-rotating Earth. In either case, scene graph nodes are used to organize the spatial and logical relationships of all elements in the scene. This also means that if a scene graph node is disabled, all of its children will not be updated or rendered either and thus will only impact the system's performance minimally.

All scene graph nodes have a unique identifier that is used to identify an individual scene graph node as well as its subcomponents. This identifier is used as part of a URI system to change the rendering parameters of scene graph nodes (Section 4.2.1) as well as to specify the parent/child relationship used to construct the scene graph in the first place. In addition, scene graph nodes can also be tagged with any number of user-defined descriptors to group these in a user interface or provide access to a group of scene graph nodes in scripts, for example setting the scaling factor for all terrestrial planets.

Figure 7 shows an example of the specification for a single scene graph node using the Lua scripting language. When loading a scene, the Lua specification of all scene graph nodes will be used to create their C++ class representation based on the provided names using a factory pattern. In this example, the `SpiceRotation` is provided by a different module (Space) from the `RenderableGlobe` (GlobeBrowsing) and do not have any dependency on each other. The `SpiceRotation` is an example of the transformation classes that are based on the NASA SPICE library [2], which is used in the majority of celestial objects and spacecraft in OpenSpace. Another example of a transformation is the `TLETranslation` which utilizes two-line elements, a common representation of the six Keplerian elements used to describe a closed orbit; a representation that is computationally more efficient than using SPICE and thus used for satellites and space debris.

```
local jupiter = {
  Identifier = "Jupiter",
  Parent = "JupiterBarycenter",
  Transform = {
    Rotation = {
      Type = "SpiceRotation",
      SourceFrame = "IAU_JUPITER",
      DestinationFrame = "GALACTIC"
    }
  },
  Renderable = {
    Type = "RenderableGlobe",
    Radii = { 71492000, 71492000, 66854000 },
    SegmentsPerPatch = 64,
    Layers = {
      ColorLayers = {
        {
          Enabled = true,
          Identifier = "Texture",
          FilePath = textures .. "/jupiter.jpg"
        }
      }
    }
  },
  Tag = { "planet_solarSystem", "planet_giants" },
  GUI = { Path = "/Solar System/Planets/Jupiter" }
}
openspace.addSceneGraphNode(jupiter);
```

Fig. 7. Specification of a scene graph node. A unique identifier for each scene graph node is used to explicitly specify the parentage of each node. The flexibility of Lua enables both the explicit definition of scene graph nodes as shown here, but also to create these programmatically.

4.1.3 Asset System

While the Module system described above provides code customization, it is desirable to provide a flexible means of setting up scenes and settings for non-programmers. This functionality enables user to create content based on new data sets using existing rendering techniques.

To enable integration of heterogeneous data into the system and to provide a flexible means for setting up scenes and settings, OpenSpace provides a subsystem for declaring modular components of data and configuration. Such components are called *Assets* and are manifested as Lua scripts that provide methods for its initialization and deinitialization. Assets may declare dependencies towards other assets and thus create a directed acyclic graph that starts with an implicitly generated root asset. Assets can request both local and external resources, which are fetched from the internet. One type of external resource uses a custom server that uses version numbering so that the data sets only need to be downloaded once and are then cached locally. This mechanism enables support for delivering updates to users easily by increasing the locally requested version number and enables forcing a download of the latest data upon start-up regardless of version numbering. This is useful when loading data from services that are continuously updated such as the most current satellite data from CelesTrak [23].

In OpenSpace, a scene is created by initializing the complete acyclic graph of a root level asset and its dependencies recursively. The system guarantees that all dependencies are initialized in topological order and that any resources residing on external servers are downloaded before initialization can begin. As an example, two independent assets consisting of a planet and a spacecraft may both reference a third asset that is responsible for downloading a common set of SPICE kernels. Whenever a new revision of the SPICE kernels is made available, the common reference can be changed, which propagates to the other assets. This increases the reusability of components.

Assets have access to Lua functions defined by the `OpenSpace-Core` and the modules. For example, assets may use the Lua functions `addSceneGraphNode` and `globebrowsing.addLayer` to create new scene graph nodes or add new virtual textures to a planetary surface respectively, use `bindKey` to create a new keyboard shortcut that is available during a session, or `setPropertyValue` to influence the rendering parameters of the system (Section 4.2.1).

4.1.4 Simple Graphics Cluster Toolkit

A major design goal of OpenSpace is to support seamless execution of the same application on a variety of different display devices. This can be achieved through the introduction of an abstract windowing layer that hides the concrete rendering setup, for example whether the scene is rendered stereoscopically, from the rendering engine. In OpenSpace, this is achieved through an API defined by `OpenSpace-Core` that is implemented by a concrete windowing framework. The default windowing framework included with OpenSpace is the Simple Graphics Cluster Toolkit (SGCT) developed at Linköping University [11]. It is a layer on top of the widely used GLFW library used to create and manage windows and rendering contexts. SGCT also provides the

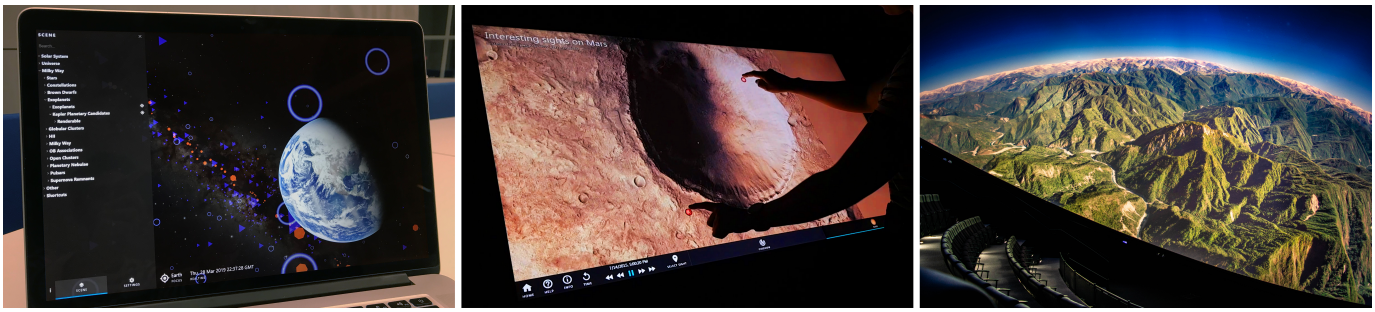

Fig. 8. A selection of display devices that are supported by OpenSpace use of the Simple Graphics Cluster Toolkit. Regular laptop and desktop systems with arbitrary number of displays (a), large-scale touch tables (b), and planetarium environments (c). In all three cases, the same executable is used with different configuration files to generate the images.

ability to synchronize data in a clustered environment using a network interface. Additionally, it provides support for performing the necessary computations to generate stereoscopic viewpoints, tiled display walls, as well as many other output rendering formats. SGCT also provides the ability to warp the output, thus enabling the support for non-planar projection methods, such as fisheye projections or spherical mirror projections. These projections are implemented by performing multiple render passes to create the required parts of a cube bounding box. These images are then reprojected as a post-processing step to create the desired output format, which can be a 180 degree fisheye projection, an equirectangular projection or any other custom format. This approach provides great flexibility as new output methods can be added easily.

In the case of SGCT's use for multi-pipeline display systems, each image generator is running the same OpenSpace application, but is rendering a different viewpoint as defined in the windowing configuration. A master node is used to process user input and modify the underlying state, which is then replicated to all connected image generators.

This solution makes OpenSpace flexible and solves the challenge of providing a visualization tool for many different presentation venues. This approach follows the "Compile once, run anywhere" paradigm as the system should not be aware of the display type that is used.

SGCT also provides built-in support for frame locking, which is a essential technique when using a mosaic projector tiling, in which different computers are responsible to render different elements of the mosaic. These setups are very common in power walls or planetarium environments. In those cases, if the timings of the projectors' frame output are not synchronized, tearing can occur on the border between two displays. For professional graphics hardware, hardware synchronization is available which forces the buffer swap to happen at the same time on the graphics driver level. For non-professional hardware, SGCT provides the ability to emulate this using light-weight network packages that are able to perform the same task.

It is possible to use other windowing frameworks. As an example, OpenSpace has been used with the MinVR windowing framework that has a stronger support for virtual reality hardware than SGCT [25, 26].

4.2 Interaction

A robust and flexible interaction scheme provided by a diverse set of input devices is vital to an effective astrographics tool. OpenSpace also supports a variety of interaction devices, among which are keyboard and mouse, joysticks, and gamepads. Additionally, the `OpenSpace-Core` defines a flexible API that can be used to add new interaction devices.

The main interaction facility is provided by the execution of Lua scripts. This defines a single common interface that makes it easy to serialize the user's interaction and thus enable reproducibility and replicability of the input (Section 4.3 and Section 4.4). However, as there is overhead when dealing with executing textual representations of scripts, the camera and global time in OpenSpace are typically handled without the use of the scripting system since these rendering parameters are expected to change potentially every frame.

4.2.1 Property System

User-changeable settings in OpenSpace are wrapped in a concept called a `Property`. Among the elements added in properties are an identifier that is unique to the owner of the property, a user-facing name and description, and a visibility setting that determines whether a property is user-facing, developer-facing, or an advanced feature.

Properties are implemented as an extension of class member variables and is thus strongly typed. Examples of properties include `IntProperty`, `Vec4Property`, `TriggerProperty`, and `StringProperty`. A property includes metadata that is used to automatically generate information for the user interfaces and the ability to access the properties from scripts (Section 4.2.2).

If the property represents a numeric value, there are options for an admissible minimum and maximum value and step value. This metadata is used by the graphical user interface to display the correct type of a component and provide the user with a reasonable interface. Additionally, properties can have *views* that ought to change the behavior of user interfaces when displaying the properties in a user interface. For example, the `Vec4Property` has a view option `Color` that is set by the developer for `Vec4Property`s that semantically represent a color and the user interface can use that information to display a color picker instead of individual numeric values. However, the user interface is free to ignore this metadata depending on the use case, for example a developer interface might want to disable all checks about minimum and maximum values.

`Property`s are owned by `PropertyOwners`, which, in turn, can be owned by other `PropertyOwner`s. This organization leads to a tree structure, providing a unique identifier for each `Property`, that can be used by scripts to get and set property values. Scene graph nodes and its subcomponents described in Section 4.1.2 are all subclasses of `PropertyOwner`, thus contributing to this unique identifier.

In addition to using the property types provided by the `OpenSpace-Core`, modules can create their own property types that are accessible for classes within that module and its dependent modules. For example, the `volume` module defines a `TransferFunctionProperty` that encapsulates a transfer function used by a volume renderer.

4.2.2 Scripting

Scripting is a fundamental aspect of OpenSpace's interaction design. Apart from being the only way to modify `Property`'s from a user interface, scripting also provides access methods to change the scene graph structure, load and unload assets at runtime, defining keyboard shortcuts, change rendering parameters, providing input mapping for interaction devices, provide a programmatic interface to the camera and the time, and much more. One benefit is the achievement of a greater level of flexibility that a full programming language provides; by treating configuration as code, rather than data, it empowers a knowledgeable user to customize an OpenSpace session to a much greater extent. A drawback of this system compared to a more declarative approach is that it impossible to validate configurations before executing them as a script can contain arbitrary code that is executed by

OpenSpace at start-up time. Additionally, writing a declarative-style text file for the configuration would be more intuitive for builders that do not have experience using a programming language.

Modules can create their own script functions either in C++ or as a Lua library that are placed in their own namespace. For the end user, the only visible distinction between scripts defined in the core and module-defined scripts is this namespacing.

Properties are set through the `setPropertyValue` Lua function, which requires the property's unique identifier as a parameter. Additionally, the function has support for regular expressions that can match parts of the property's unique identifier (URI). This can be used to edit a large number of property values simultaneously. For example, the script `openspace.setPropertyValue('Scene.*.Renderable.Enabled', false)` would disable all objects in the scene graph simultaneously.

4.2.3 Network API

Interoperability with other software systems is important for many astrographics use cases, including integration with other visualization tools and custom user interfaces for installations in exhibitions.

Based on the desire to support a diverse set of tools and languages, OpenSpace provides a network interface, that can be configured to accept connections from other applications using both TCP sockets and WebSockets. The protocol is based on JSON and the notion of `Topics`, which are independent communication channels, allowing multiple concurrent and asynchronous requests to the OpenSpace main application. For example, subscribing to a property value can be achieved by initializing a `Topic` of the type `Subscription` and specifying the property's unique identifier. Whenever the `Property` value updates, the client receives a message within that `Topic`, specifying the new value of the `Property`. Using a `Topic` of the type `LuaScript`, it is possible execute and acquire the return values of custom lua scripts. OpenSpace modules may add additional `Topic` types to provide additional sub-protocols for interacting with module specific functionality.

4.2.4 Navigation

The default interaction mode in OpenSpace is based on the concept of an orbital navigation mode. In this mode, the user designates an arbitrary scene graph node as the *Focus Node*, which results in all interactions being performed relative to that node. The possible interactions include a circular (orbital) movement around the object, translation along the line connecting the camera and the center of the focus node, a roll rotation around the same connecting axis, and an additional local camera rotation that enables panning movements. The camera model also supports optional friction modes for each of these interaction components that slow down the camera over a period of a few seconds if no continuous interaction is performed.

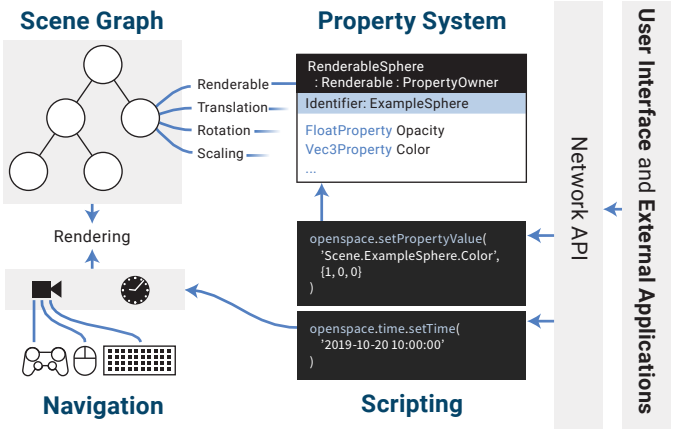

Fig. 9. The scene graph is populated by instantiating classes provided in `Modules`. Scene graph objects expose properties, that are changeable through scripting, which in turn is exposed via a Network API. Navigation is controlled directly through input devices or using the Network API.

As the camera position is defined relative to the focus node, it will follow its movements. Otherwise, it would be cumbersome when focusing on a fast moving object, such as Earth in its orbit around the Sun. In addition to the relative position of the camera, OpenSpace also supports an automated co-rotation with the selected focus node if the camera is closer to the node than a user-defined distance threshold. If the distance is below the threshold, the object's rotation will also be applied to the camera, otherwise it will rotate underneath the camera. This enables the user to place the camera on the surface of a planetary body and see the movements of other objects when playing back time.

All interaction components are automatically scaled exponentially by the distance from the camera to the focus node. The exponential scaling addresses the challenge of large scales that are present in astrographics. For example, when navigating the surface of a planetary body, the user expects the camera to move in the velocity ranges of meters per second, whereas when navigating the Sloan Digital Sky Survey of galaxies, we need to be able to move in the velocity ranges of million light-years per second. The exponential factor and a multiplicative factor, can be changed by the user to customize the desired input sensitivity.

Modules can define their own interaction methods, such as functionality that specializes the interaction for multi-touch-capable input devices [6] or devices using accelerometers, such as phones or tablets.

Temporal Navigation Every object in OpenSpace is using a single unified time provided by the `OpenSpace-Core` to synchronize data sets in order to help with contextualization. The *Delta Time* determines the speed at which time advances in the simulation in the units of $\frac{\text{simulation time}}{\text{wall clock time}}$. The user can interactively change the delta time to inspect events that occur at different timescales, for example seconds in the case of rover motions on Mars, or tens of thousands of years in the case of stellar movement.

The time in OpenSpace is represented as the number of seconds past the 2000-01-01 epoch and is stored as a double precision floating point number. Although this poses limitations on the achievable possible precision (double machine ε is about $2 \cdot 10^{-16}$), this still provides nanosecond accuracy for contemporary data sets. The range of representable numbers in double precision format is $\approx 10^{300}$ years, which is sufficient for astrographics data.

4.2.5 User Interface

The OpenSpace system provides a built-in graphical user interface (GUI) constructed with the Chromium Embedded Framework (CEF), which is a library used to render webpages [16]. The user interface webpage uses the Facebook React and Redux libraries [13, 14] for a dynamic and responsive GUI. It is served by a local stand-alone webserver and rendered in OpenSpace in a separate module using CEF. The user interface communicates with the running OpenSpace instance on the local machine using a WebSocket connection (Section 4.2.3).

Typically, the GUI is rendered on top of the visual output of the OpenSpace scene on the user's desktop. However, it is also possible to decouple the GUI from the OpenSpace instance by using an external browser instead. This is useful if the main rendering window is shown in a planetarium dome and the user interface would clutter the view. Furthermore, it enables access to the GUI from any device with a web browser, such as a tablet or smart phone.

This system is designed such that the rendered web page is determined by the scene that is loaded. This enables the use of specialized user interfaces for scenes and also enables the rapid prototyping of user interfaces without the need to recompile the OpenSpace application.

The development of OpenSpace started with a native, intermediate-mode graphical user interface using the `ImGUI` C++ library. OpenSpace migrated to the Chromium Embedded Framework and React/Redux to increase the flexibility that comes with an easily editable user interface that does not require a recompilation and thus enables a faster iteration time. This enables a more rapid prototyping of new interface components. The separation of the underlying rendering component of OpenSpace from the generation of the user interface also reduces the required knowledge about the rendering that an interface designer has to possess in order to create new interface components.

4.3 Collaborative Experiences / AstroCasting

One of the major potential benefits of astrographics in general and for OpenSpace in particular is the increase in potential collaborations across geographically distributed locations. There are two kinds of collaborations to be considered.

First, a domain expert speaker wishing to give a public presentation about their topic of interest using their own data. Instead of presenting to a single venue at a time, and thus limiting the number of people that can participate and learn from the presentation, it would be beneficial to include a greater number of other venues in this presentation. This includes streaming the audio and video feed of the presenter, as well as synchronizing the rendering that each venue is experiencing.

The second type of collaborative experiences involve groups of scientists that desire to share discovery sessions that include new data sets or new findings. This technique has the potential to increase the efficiency of these sessions as it allows for an immediate dissemination of the data sets. For this to be effective, it is essential to be able to pass session control between the different partners.

The synchronization of the rendering in OpenSpace is achieved by streaming the state changes to the camera and time, as well as all executed Lua scripts. This approach is superior to streaming video in two ways: 1) the required bandwidth to stream rendered video is much greater than streaming the state of an application like OpenSpace and 2) it is possible to hand-off control of the session to a client site increasing the collaborative aspect of the event.

A stand-alone server application called *Wormhole* is serving as the central hub to which individual OpenSpace instances connect. This can be password protected or open to the world. In the presentation case described above, the connecting partners might be planetariums or other public venues, but this also allows users on their own home computer to connect and join the collaborative session remotely. Each joint session has one *host* at a time and an arbitrary number of *peers*. All camera movements, time changes, and all Lua scripts that are executed by the host are sent to the *Wormhole* server and then automatically distributed to the connected peers that will update their own local OpenSpace instance according to their local display setup. This makes it possible to control various display devices such as spherical display devices in planetariums, power walls, warped projections, and home computers at the same time offering tremendous flexibility. If the creator of the joint sessions has enabled shared control, it is possible for a peer, knowing the correct password, to seemlessly assume control over the session becoming the host and continue the exploration.

4.4 Session Provenance

We desire the ability to record and replay sessions to foster reproducibility, distribute sessions between planetariums, and address the challenge of collaboration between domain scientists using OpenSpace. While some of these use cases could be solved by recording regular videos of an OpenSpace session, there are other cases when a more elaborate solution is required. For example, it is desirable to be able to replay a session with a different set of configurations or on a different projection system. Furthermore, it is useful to be able to stop the playback, interact with the system, and resume the playback.

To address these scenarios, OpenSpace contains a subsystem that manages full session provenance. When enabled, the system records the camera position and orientation as well as the current time and delta time each frame. In addition, each executed Lua script is recorded. As described above, Lua scripts are the only way for the user to change the state of the OpenSpace system and, thus, it is guaranteed that this mechanism captures all possible changes during an OpenSpace session which can then be saved to a local file. When the provenance file is later played back, the session will be repeated automatically with exactly the same behavior as during the initial recording.

Capturing the state of the system has benefits over similar approaches. Capturing individual frames and storing the session as a video would not have the ability to stop the playback and interact with the system *a posteriori*; capturing the raw mouse and keyboard input would make it cumbersome to redistribute the resulting files to a different display setup, as the input might depend on the type of display and its resolution or the available input modalities and manually executed Lua scripts would be missed. Instead, by capturing the entire provenance, it becomes possible to distribute session files among researchers wishing to collaborate asynchronously or playback planetarium presentations as semi-interactive shows. Additionally, it is also possible to save a distributed collaborative session as described in the previous section to a provenance file that can be stored for prosperity.

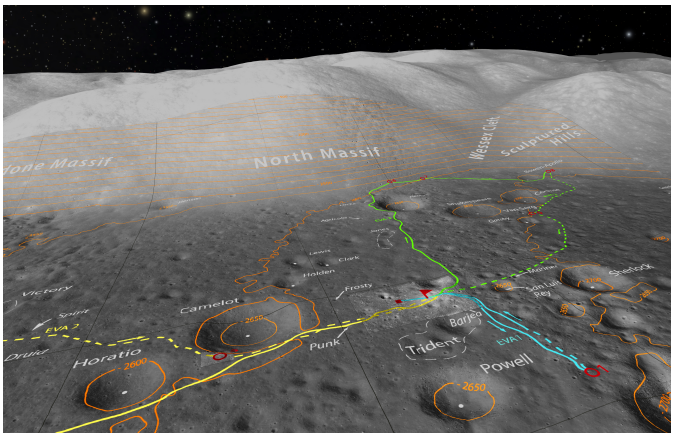

Fig. 10. Visualization of the excursions of the Apollo 17 astronauts using the Lunar Roving Vehicle in the context of satellite imagery of the landing site from Lunar Reconnaissance Orbiter.

4.5 Performance

The performance measurements that are reported have been conducted on a 2.9 GHz Intel i9 CPU with an nVidia Geforce 2080 graphics card rendering a 3840×2160 resolution output image and the framerates are averaged over several hundred frames. All rendering parameters, including those which impact performance, are the same used for creating their respective images.

- Apollo capsule (Figure 1, left): 46 fps.
- Sloan Digital Sky Survey (Figure 1, right): 67 fps.
- Coronal Mass Ejection (Figure 3): 55 fps.
- New Horizons (Figure 4): 81 fps.
- Apollo 17 Landing site (Figure 10): 47 fps.

5 Discussion and Utilization

This section discusses how OpenSpace is utilized in the context of the use cases for astrographics described in Section 1.1 and how its design contributes to these usage scenarios.

Tool in astronomy research OpenSpace is being used by the Community Coordinated Modeling Center at NASA Goddard to visualize and analyze space weather simulations [8]. Figure 3 shows the Bastille day coronal mass ejection visualized using time-varying fieldlines depicting the induced magnetic field and a time-varying volumetric rendering of the plasma density. Additionally, OpenSpace has been used to render the Gaia data set [10] to a meeting of leading Gaia researchers and was used in a large collaborative session investigating this data (Figure 2). Key OpenSpace components supporting these use cases are:

- The `Asset` system described in Section 4.1.3 that enables the customization at start-up of an OpenSpace session needed to support dynamic data.
- The use of a JSON-based interface for the networking API creates the possibility to use other programming languages, mostly Python and IDL in the case of heliophysics experts, to connect to OpenSpace and provide information over the socket interface.
- The definition of a fixed data format for stellar measurements, in this case the SPECK format, made it possible for a selected number of researchers to provide their own custom data sets on short notice and include these in the collaborative session.

Tool for interactive planetarium shows OpenSpace has frequently been used for a number of years in an increasing number of planetariums around the world. Among many other topics, this includes presentations of the large scale structure of the Milky Way and universe using the Digital Universe data catalog [1]. Some of components of OpenSpace enabling this are:

- The Simple Graphics Cluster Toolkit which supports a number of different display systems. The ease of creating SGCT configuration files enabled the installation of OpenSpace on currently 20 informal science institutions around the world. The definition of a common interchange format in SGCT also fostered collaborations with all major planetarium hardware vendors.
- The flexibility of the `Asset` system and the ability to dynamically load new content which has enabled many of the partners to create assets without the help of the core development team. Figure 10 shows one result of this flexibility as a visualization of the landing site of Apollo 17.
- The flexible user interface (Section 4.2.5) which makes it possible to expose the software to a greater number of facilitators at planetariums as well as novice users in the general public.

Tool for mission planning and communication OpenSpace was used in several linked planetariums to show the New Horizons fly-by in real-time [5, 7] which was accompanied by descriptions from scientists on the mission team as the fly-by was occurring. The same techniques also enabled the visualization of the Rosetta [5] and OSIRIS-REx [6] missions. The enabling OpenSpace features include:

- AstroCasting (Section 4.3), which has enabled a number of successful public presentations that included multiple planetariums, as well as live streams to video streaming services.
- Local recording using the session provenance (Section 4.4) and the play back at a later time which creates the ability to include geographically distributed locations in different time zones.

Over the last four years, OpenSpace has been used in a large number of public presentations, which included local activities, presentations using the Astrocasting system, as well as productions for video streaming services. Through all these modalities, the system has reached over 500 000 members of the public, showing the impact that open source software can have on exposing these audiences to scientific data. Additionally, there has been an informal session with the Mars 2020 Science Definition Team inspecting CRISM data for the potential landing sites of the Mars 2020 rover. This collaboration was only possible due to the reliance on well-known third-party libraries which acted as a common interface to the mission science team.

6 Future work

As with any other long-term developed software system, the development will never reach the point where no more future work is left to be done and OpenSpace is no exception to this. However, from our experience of conducting public presentations in astrographics scenarios, we were able to distill potential additions to a number of features that would have the greatest impact on similar software systems.

Offline rendering. While OpenSpace is designed first and foremost for real-time usage, there are many situations where the creation of a high-resolution video is desirable. For example, when the required resolutions are not achievable yet in real-time or for producing movies to be shown in a planetarium without the hardware to operate an interactive software. To be able to support this, we are planning to extend the session provenance and rendering components of OpenSpace to be able to export individual frames at high-resolution when playing back a session recording. This technique also requires the extension of the rendering API for custom renderable objects to be able to wait until all available data has been downloaded before creating an image. With this method, the domain expert can record an OpenSpace session at low resolution at interactive frame rates on their own laptop and use the session provenance file to render the same sequence at a higher resolution *a posteriori*.

Dynamic Modules. Currently, all modules for a specific OpenSpace application have to be selected at compile time. This poses a limitation on *Developers* as they need to compile the entirety of all modules, `OpenSpace-Core`, and the application every time they want to add a new module. To be able to engage a greater number of developers and reduce the time required to add a new module, it would be beneficial to be able to refactor this using a plug-in architecture instead, where modules can be loaded and unloaded at runtime. This entails that developers can create new modules without requiring the source code other than their own module, which would, in turn, improve the efficiency of developing new modules.

OpenSpace Content Builder. To mitigate the complexity of using the Lua scripting language to specify assets, we are planning to create a graphical software tool that helps *Builders* to create new assets quickly. By analyzing which modules are currently available for an OpenSpace instance, it is possible to automatically create lists of available classes and provide a graph-based user interface to the user to drag and drop new asset trees and then export these as Lua-formatted assets. In addition to being able to quickly generate new assets, exporting them as Lua also enables the builder to further customize the assets after they have been created to retain the full ability to customize assets.

Content database. Utilizing the capabilities of a content builder tool, we expect builders all around the world to quickly create a large number of potentially useful assets that they would want to share with other users of OpenSpace. Similar sharing services exist for other software systems, such as Celestia, but using the ability to modify the scene graph at runtime enables OpenSpace to expand on these databases by being able to automatically download an asset and directly integrate it, so that the time between selection of an asset and inclusion in a public presentation would be minimal, furthering the ultimate goal of reducing the time between discovery and presentation.

Other domains. While OpenSpace was developed for astrographics, the OpenSpace system is flexible. It would be interesting to investigate how this could be applied to other domains with similar visualization challenges, such as molecular biology.

7 Conclusions

In this paper we have presented the OpenSpace system, an open source initiative in the area of interactive astrographics. The ambitious goal for OpenSpace is to serve as a platform for three different but connected areas: 1) Science communication 2) Research in astronomy and space science 3) Research in visualization. The underpinning idea is the confluence of exploratory and explanatory visualization and in particular the use of research data in science communication. We have described typical use cases of the system, each showing how OpenSpace meets the challenging mission of providing interactive visualization in a range of contexts, including planetarium shows, interactive installations at museums and science centers as well as supporting scientific discovery workflows. Over the years a participatory design process has been used in science to solution cycles to meet the challenges posed by interactive astrographics. The system architecture is modular and configurable, allowing developers, scientists, and science communicators to tailor OpenSpace to their needs, import their own data sets and share interactive sessions. The resulting OpenSpace system is a vehicle for the introduction of the next generation of interactive and data driven use of visualization in astronomy and space exploration.

Acknowledgments

This work was supported by NASA Cooperative Agreement Notice under grant NNX16AB93A, the Knut & Alice Wallenberg Foundation through the WISDOME project, VR grant number 2015-05462, the Swedish e-Science Research Center, the Moore-Sloan Data Science Environment at NYU, NSF awards: CNS-1229185, CCF-1533564, CNS-1544753, CNS-1626098, CNS-1730396, CNS-1828576. The presented software system source code is available at `https://github.com/OpenSpace/OpenSpace`.

REFERENCES

[1] B. Abbott. The digital universe guide for partiview, 2006.

[2] C. H. Acton Jr. Ancillary Data Services of NASA's Navigation and Ancillary Information Facility. *Planetary and Space Science*, 44(1):65–70, 1996. doi: 10.1016/0032-0633(95)00107-7

[3] E. Axelsson, J. Costa, C. T. Silva, C. Emmart, A. Bock, and A. Ynnerman. Dynamic Scene Graph: Enabling Scaling, Positioning, and Navigation in the Universe. *Computer Graphics Forum*, 36(3):459–468, 2017. doi: 10.1111/cgf.13202

[4] K. Bladin, E. Axelsson, E. Broberg, C. Emmart, P. Ljung, A. Bock, and A. Ynnerman. Globe Browsing: Contextualized Spatio-Temporal Planetary Surface Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):802–811, 2017. doi: 10.1109/TVCG.2017.2743958

[5] A. Bock, C. Emmart, M. Kuznetsova, and A. Ynnerman. OpenSpace: Changing the Narrative of Public Disseminations in Astronomical Visualization from *What* to *How*. *IEEE Computer Graphics and Applications, Special Issue – Applied Vis*, 38(3), 2018. doi: 10.1109/MCG.2018.032421653

[6] A. Bock, C. Hansen, and A. Ynnerman. OpenSpace: Bringing NASA Missions to the Public. *IEEE Computer Graphics and Applications*, 38(5):112–118, Sep./Oct. 2018. doi: 10.1109/MCG.2018.053491735

[7] A. Bock, M. Marcinkowski, J. Kilby, C. Emmart, and A. Ynnerman. OpenSpace: Public Dissemination of Space Mission Profiles. In *Proceedings of the Scientific Visualization Conference (Poster)*, pp. 141–142. IEEE, 2015. doi: 10.1109/SciVis.2015.7429503

[8] A. Bock, A. Pembroke, M. L. Mays, L. Rastaetter, A. Ynnerman, and T. Ropinski. Visual Verification of Space Weather Ensemble Simulations. In *Proceedings of the Scientific Visualization Conference*. IEEE, 2015. doi: 10.1109/SciVis.2015.7429487

[9] M. Boylan-Kolchin, V. Springel, S. D. White, A. Jenkins, and G. Lemson. Resolving cosmic structure formation with the Millennium-II Simulation. *Monthly Notices of the Royal Astronomical Society*, 398(3):1150–1164, 2009. doi: 10.1111/j.1365-2966.2009.15191.x

[10] A. Brown, A. Vallenari, T. Prusti, J. De Bruijne, C. Babusiaux, C. Bailer-Jones, M. Biermann, D. W. Evans, L. Eyer, F. Jansen, et al. Gaia data release 2-summary of the contents and survey properties. *Astronomy & Astrophysics*, 616:A1, 2018. doi: 10.1051/0004-6361/201833051

[11] C-Research. *Simple Graphics Cluster Toolkit: http://sgct.itn.liu.se/*, 2018.

[12] Celestia. *Celestia - real-time 3D visualization of space: https://celestia.space/*, 2001-2018).

[13] R. Community. React: https://reactjs.org/, 2019.

[14] R. Community. Redux: https://redux.js.org/, 2019.

[15] Evans and Sutherland. *Digistar Planetarium Software: https://www.es.com/Digistar/*, 2018).

[16] C. E. Framework. *Chromium Embedded Framework: https://github.com/chromiumembedded/cef*, 2019.

[17] A. S. García, D. J. Roberts, T. Fernando, C. Bar, R. Wolff, J. Dodiya, W. Engelke, and A. Gerndt. A collaborative workspace architecture for strengthening collaboration among space scientists. In *2015 IEEE Aerospace Conference*, pp. 1–12. IEEE, 2015.

[18] A. Gerndt. Collaborative Rover Operations and Planetary Science Analysis System based on Distributed Remote and Interactive Virtual Environments. Final Publishable Summary Report, European Union Framework Programme 7.

[19] K. M. Gorski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. Healpix: a framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759, 2005.

[20] T. Gutschmidt. *Game Programming with Python, Lua, and Ruby*. Premier Press, 2004.

[21] R. Ierusalimschy, L. H. De Figueiredo, and W. C. Filho. Lua — An extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[22] JPL. Nasa eyes: https://eyes.jpl.nasa.gov, 2019.

[23] T. Kelso. Celestrak: https://celestrak.com/, 2019.

[24] S. Klashed, P. Hemingsson, C. Emmart, M. Cooper, and A. Ynnerman. Uniview - Visualizing the Universe. In *Eurographics - Areas Papers*. Eurographics Association, 2010. doi: 10.2312/ega.20101005

[25] MinVR. *MinVR: https://github.com/MinVR/MinVR*, 2018.

[26] J. Novotny, M. Turner, S. Gatesy, F. Drury, P. Falkingham, and D. Laidlaw. Developing virtual reality visualizations for unsteady flow analysis of dinosaur track formation using scientic sketching. *Transactions on Visualization and Computer Graphics*, pp. 2145–2154, 2019.

[27] D. J. Roberts, A. S. Garcia, J. Dodiya, R. Wolff, A. J. Fairchild, and T. Fernando. Collaborative telepresence workspaces for space operation and science. In *Virtual Reality*, pp. 275–276. IEEE, 2015.

[28] 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

[29] Skyskan. *DigitalSky 2 Software for Definiti Theaters: https://www.skyskan.com/products/ds*, 2018).

[30] A. A. Society. *World Wide Telescope: http://www.worldwidetelescope.org/home*, 2018.

[31] Stellarium. *Stellarium Astronomy Software: https://stellarium.org/*, 2018.

[32] F. Warmerdam. The Geospatial Data Abstraction Library. In *Open Source Approaches in Spatial Data Handling*, pp. 87–104. Springer, 2008.

[33] A. Ynnerman, J. Löwgren, and L. Tibell. Exploranation: A new science communication paradigm. *IEEE Computer Graphics and Applications*, 38(3):13–20, 2018. doi: 10.1109/MCG.2018.032421649