AIAA SciTech Forum
January 3-7, 2022, San Diego, CA & Virtual
AIAA SCITECH 2022 Forum

10.2514/6.2022-1433

IRIS: High-fidelity Perception Sensor Modeling for Closed-Loop Planetary Simulations

Carolina Aiazzi*, Aaron Gaut† and Abhinandan Jain‡
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA., 91109, USA

Aaron Young§ and Asher Elmquist¶
University of Wisconsin-Madison, WI, 53706, USA

Perception plays a key role for autonomous and semi-autonomous planetary exploration vehicles. For instance, landers can use computer vision techniques for identifying safe landing locations, aerial vehicles use cameras as navigation sensors, and planetary rovers use them for localization and hazard detection. Engineering simulations of such systems require the accurate modeling of perception and vision sensors for simulating autonomy scenarios. In addition, the modeling of sensors for landers, aerial and ground vehicles requires the ability to handle large and high-resolution terrain data, the accurate modeling of illumination, hi-fidelity rendering via ray/path tracing, and the modeling of sensor characteristics. Vision sensor models simulate sensor reality by using physics principles to model the interaction of light and objects. High frame rate performance is highly desirable for in-the-loop simulations involving vehicle dynamics and control software. In this paper we describe a new sensor modeling capability called *Inter-planetary Rendering for Imaging and Sensors (IRIS)* that meets these requirements for the real-time and high-fidelity closed-loop simulation of vision sensors for planetary aerospace and robotics applications.

Nomenclature

BRDF	=	Bi-directional Reflectance Distribution Function
PBR	=	Physically Based Rendering
w	=	single scattering albedo
$B_{0,SH}$	=	amplitude of shadow-hiding opposition effect
h_{SH}	=	width of shadow-hiding opposition effect
$B_{0,BC}$	=	amplitude of coherent backscatter opposition effect
h_{BC}	=	width of coherent backscatter opposition effect
g	=	asymmetry factor
θ	=	macroscopic roughness angle
e	=	emission angle
i	=	incident angle
μ	=	cosine of e
μ_0	=	cosine of i
μ_e	=	cosine of e considering the macroscopic roughness angle
μ_{0e}	=	cosine of i considering the macroscopic roughness angle

*Robotics Technologist, Mobility and Robotic Systems Section.
†Robotics Systems Engineer, Mobility and Robotic Systems Section.
‡Senior Research Scientist, Mobility and Robotic Systems Section.
§Mechanical Engineering Undergraduate Student
¶Mechanical Engineering PhD Candidate.

I. Introduction

Closed-loop flight dynamics simulations play a critical role in the development of space and robotics flight systems. Such high-fidelity simulators for aerospace and robotics applications typically include rigid and flexible body dynamics models of the vehicles, together with accurate models of their actuator and sensors along with high quality models of the environment. Such simulators are typically expected to perform in real-time, and even faster than real-time performance for closing the loop with onboard flight software. High-fidelity is especially important for engineering quality simulation. Onboard autonomy is playing an increasingly important role in the design and operation of such space flight systems. Recently, the NASA Ingenuity Mars helicopter successfully demonstrated semi-autonomous navigation during flight on Mars, and the Mars 2020 Perseverance rover is carrying out autonomous traverses on Mars. Autonomous platforms typically rely on perceptions sensors such as cameras and lidars and sensor processing algorithms to provide the necessary data required by the onboard autonomy algorithms. The terrain-relative navigation (TRN) system for Mars 2020 lander relied on imaging data during descent to assess hazard risks and select the landing site. The reliance on autonomy is expected to grow significantly with increasing need to operate space robotic platforms, such as in Figure 1, in uncertain and unstructured environments. Supporting such goals are ongoing technology development and flight projects to demonstrate multi-agent autonomous operation of rovers on the moon, autonomous exploration and sampling of Ocean Worlds such as Europa and Enceladus and autonomous aerial platforms for Venus and Mars. Such needs are providing an impetus for advancing flight dynamics simulations to develop additional capabilities required for closed-loop simulations for autonomous platforms.

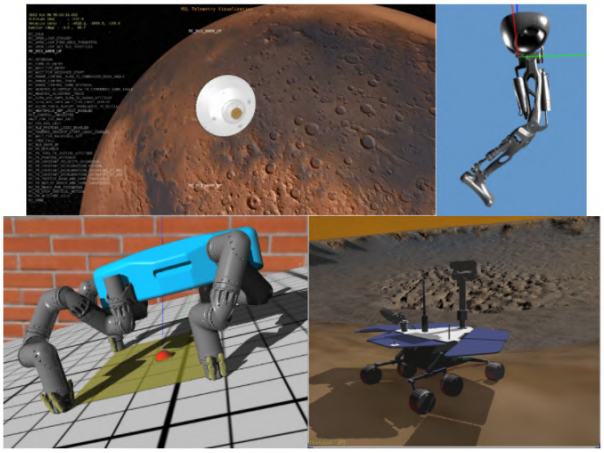


Fig. 1 Examples of aerospace and robotics systems relying on sensors include landers, ground vehicles and robots.

The addition of autonomy requires simulators to support the closed-loop development and testing of onboard guidance, navigation and control (GNC) together with machine vision algorithms that are a part of the autonomy stack for aerospace and robotic platforms. In particular, this requires good fidelity and performance of perception sensor models which play a key role in generating the data required for the onboard reasoning, planning, and control software. Such sensor simulations require the proper modeling of the environment's properties such as planetary surface topography, lighting and illumination models, as well as sensor nominal and non-ideal characteristics (eg. noise, motion blur). Landers and aerial platforms can traverse over very large areas, while at the same time requiring high-resolution detail close to landing sites. Simulation solutions need to scale up to handle such large dynamic range in size and resolution of terrain data sets. Modeling the illumination of planetary surfaces can be complex and needs to take into account shadowing and non-homogeneous reflectance and scattering properties to simulate phenomena such as *opposition effects*. Such sensor models are required to integrate with and operate within vehicle dynamics simulations. In addition, closed-loop engineering simulations require sufficient performance speed from the sensor models for effective evaluation and testing of the autonomy scenarios. We summarize below some of the key challenges involved in modeling perception sensors for meeting autonomous system simulation needs.

Light sources: An important and often the primary lighting source is the sun. Ephemerides data is required to properly position such a sun lighting source. Other planetary bodies in the vicinity may also serve as additional light sources. Furthermore, artificial illumination sources such as spot lights may also need to be modeled. Accurate modeling of shadows can be challenging, especially in low lighting conditions and from having to deal with multiple paths.

Material reflectance properties: The appearance of bodies is determined by their reflectance and opacity material attributes. The widely used *Phong* shading model [1] defines reflectance properties in terms of diffuse, ambient, and specular color properties of materials. More advanced models such as *Physically Based Rendering (PBR)* models provide a more sophisticated way to model the appearance of body surfaces with properties such as metalness, roughness, reflectivity, translucency, etc. Support for these different material types during rendering is required.

Planetary surface illumination: Planetary surfaces often are made up of regolith. There are several proposed models for illumination of such regolith and other planetary surfaces including ones known as Oren-Nayar, Hapke etc. The Hapke model is one of the few that is able to successfully model opposition effects.

Perception sensors: The onboard autonomy typically relies on sensor data - usually from perception sensors such as cameras and LIDARs. The realistic modeling of autonomy perception sensors requires modeling their nominal behavior as well as their various sensor distortions and noise sources as specified via calibration parameters. For cameras, the nominal properties consist of field of view, as well as their position and orientation. Camera non-idealities can include lens distortion (CAHVORE), noise, blur, vignetting, etc. For LIDARs, the nominal characteristics are the sensor geometry, as well as the scan pattern. Non-idealities can consist of return signal strength based on the reflectivity and absorption properties of the material, ray divergence and multiple ray bounces. Vehicle platforms will often host multiple sensors of different types. A sensor modeling architecture in general needs to be flexible enough to evolve to accommodate the arrival of new types of sensors.

Large and high-resolution terrain models: For planetary missions, the environment can require the modeling of very large terrain data sets spanning thousands of kilometers, while at the same time supporting high-resolution terrain data for landers and surface and near-surface platforms. The combination of large size and high resolution makes the modeling of such terrain data sets very challenging. A level-of-detail (LOD) process is needed to manage the size of such models. Insufficient surface detail needs to be procedurally enhanced to meet the resolution requirements for machine vision algorithms. In addition, there is a need to model the visual appearance of wheel tracks for ground vehicles since they are typically in the field of view of sensors. Furthermore, there is a need to include sky and star maps in the environment models.

Large spatial distances: Another significant issue when working with perception sensor models for space applications can be the large distances involved and the significant loss in precision in computing relative distances. While the distance to the sun and planetary radii are in the range of thousands to millions of kilometers, vehicle and planetary surface features are at the meter scale level. Special techniques are needed to avoid the significant loss in precision leading to artificial jitter in the sensor models. This situation is aggravated by the fact that most rendering software use single precision computations.

Integration within simulation: An important goal is the use of sensors within system-level simulations for autonomous aerospace and robotics platform simulations with dynamics and control software in the loop. Thus, general purpose interfaces need to be available to: allow external inputs for the creation of scenes consisting of vehicles and environments for desired scenarios, allow updating of rendering scene transforms and frames at run-time, allow changes to the scene such as addition, removal and reconfiguration of scene objects. Furthermore, interfaces must be available for returning data back to feed into the sensor simulation modules. Sensors can be mounted on articulated platforms on a vehicle and sensor models need to accommodate such motions. A desirable goal is to provide an interface that is usable by analysts for setting up complex and a variety of scenarios with minimal demands on rendering expertise.

Real-time performance: For effective use in closed-loop simulations, it is important that sensor simulators provide multiple frames per second performance speed to enable closed-loop simulation of scenarios. This is important for supporting scenarios and analyses that can extend over significant time intervals, as well as for Monte-Carlo simulations that are commonly used in these contexts. Such large scale simulations are often deployed on large-scale cloud computing servers. As such, the ability to run such simulations in *headless* mode (i.e. without onscreen graphics, despite the reliance on rendering techniques) can be important for the deployment to such computing resources and for acceptable performance.

Graphics visualization: Due to the complexity of dynamics simulations, it is very helpful to have real-time visualization of the simulated vehicle behaviour in real-time. Useful visualization features can include the ability to show frames of interest, trails, markers, frustums, LIDAR point clouds, text overlays, etc. Support for ornamental objects and various debug modes to visualize sensor footprints and other overlays can be very helpful. Support for chase views to automatically follow the vehicle along its path can also be very useful.

An attractive approach for meeting the high performance requirements for perception sensor simulations has been to make use of GPU-hardware accelerated rendering techniques. OpenGL *raster-based* graphics rendering techniques provide an important approach for carrying out such rendering with good real-time performance using GPU-accelerated hardware. GPU shaders can be used to implement additional effects (eg. shadows) that are not natively available within OpenGL. The popularity of raster based rendering is based on its speed though with marginal rendering quality and fidelity. Our previous approaches for such sensor modeling [2] have made use of such GPU-accelerated

raster-graphics rendering solutions to meet perception sensor simulation needs. However, achieving adequate fidelity with raster-graphics has been challenging. Even modeling shadows properly via rasterization methods is difficult and requires complex and fragile techniques.

An approach for higher fidelity sensor modeling is to make use of *ray-tracing* and *path-tracing* rendering techniques. These techniques more naturally capture illumination physics and lead to significantly superior rendering fidelity. For instance, rendering shadows comes naturally to ray tracing approaches, while requiring special purpose and fragile techniques such as shadow mapping when using rasterization approaches. *Blender* [3] and *POV-Ray* [4] are popular tools that leverage ray tracing and are commonly used for producing photo-realistic scenes. However, these techniques suffer from large computational cost and as such are very slow and not usable for closed-loop simulations. The recent availability of GPU hardware-accelerated ray-tracing has provided a path to overcome the performance bottlenecks impacting ray-tracing approaches. In this paper we described the next-generation *Inter-planetary Rendering for Imaging and Sensors (IRIS)* perception sensor capability that uses such GPU-accelerated ray-tracing techniques for the high-fidelity simulation of perception sensors in closed-loop dynamics simulations for space platforms. IRIS builds on the CUDA-based, GPU-accelerated OptiX [5] ray tracing library for overcoming computational bottlenecks normally associated with ray tracing. IRIS allows us to carry out much higher fidelity perception sensor simulations at several frames per second as needed for aerospace and robotics closed-loop simulations. Figure 2 shows an example rendered output from IRIS.

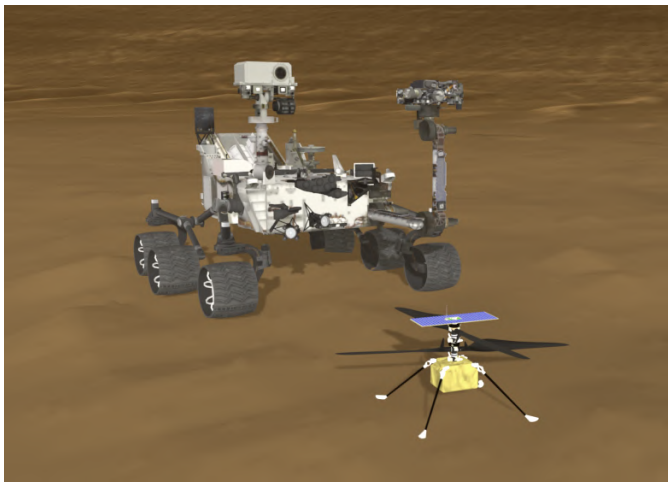


Fig. 2 An IRIS rendering of Ingenuity and Perseverance on Jezero crater using PBR materials for illumination modeling.

A. Related Work

Several illumination simulation software and tools have been developed and are accessible to the scientific and engineering community. The *PANGU (Planet and Asteroid Natural Scene Generation Utility)* software has been implemented for the visualization and simulation of planetary surfaces, using synthetic and real data [6]. It aims to help design landers that use computer vision to avoid obstacles while navigating to their landing site, produce realistic simulations of RADAR and LIDAR guidance sensors and onboard navigation cameras [7]. The images rendered can be utilized in different test setups, such as off-line, closed-loop, open-loop, and hardware-in-the-loop. Its principal focus is on the development of artificial camera images, and it can also be used to create artificial surfaces, which can represent cratered planets [8]. The renderings are done in real-time using OpenGL 4.0 and programmable GPU shaders. PANGU has been developed by the University of Dundee, supported by ESA (European Space Agency) and STFC (Science and Technology Facilities Council).

SurRender is another software package, developed by Airbus Defence and Space, to generate simulated images. The rendering method used is raytracing for non-real-time rendering, while it employs OpenGL for real-time simulation, [9]. It includes optional BRDFs, such as Lambert, Hapke, Phong, and Oren-Nayar. SurRender has its own embedded modeling language, called SuMoL, to go beyond existing models for shaders, sensors, and shapes. Similar to PANGU, it is capable of simulating different types of sensors (such as visible, LIDAR, motion blur etc.). It has been designed to manage massive data sets, called *giant textures*, which are useful to store digital elevation models (DEMs), albedo maps, and fractal texturing that creates high-quality renderings for a broad range of distances, [9]. These sensor simulation tools are able to provide either real-time or high-fidelity rendering, but not both at the same time. Both PANGU and SurRender have a client-server protocol (TCP/IP).

Another terrain simulator built on Gazebo for simulating a rover vehicle on the Moon is described in [10]. This simulator uses the raster-based Ogre3D graphics software for sensor modeling. This simulator also includes capabilities for enhancing the resolution of lunar terrain DEMs.

Relying on rendering and simulation techniques, robotic and autonomous vehicle simulation platforms have been

extensively developed for terrestrial purposes. These share several key components to planetary simulation including rigid body dynamics and sensor simulation for closed-loop testing. In the field of robotics, *Gazebo* [11, 12] has a large user base and wide breadth of sensor support. Although widely used, Gazebo is limited in it sensor and visual fidelity, relying primarily on Ogre raster graphics, with new development thrusts in progress to improve graphics support and leverage modern rendering techniques. A more modern platform is NVIDIA *Isaac Sim* [13] which leverages state-of-the-art graphics and hardware support, but detailed information on sensor support is limited due to the closed nature of the platform. Further visual fidelity, specifically for rendering cameras, can be found in on-road autonomous vehicle simulation frameworks including *AirSim* [14], *Carla* [15], NVIDIA *Drive* [16], *SynCity* [17]. These on-road framework rely heavily on high-end, game-quality rendering engines such as *Unity 3D* and *Unreal Engine*. AirSim and Carla are largely the most popular open-source, on-road simulators due to their availability and graphics support including support for complex vegetation and expansion to game mechanics such as animated objects, humans, and textures. However, due to their reliance on game technology, the fidelity of the physics is limited to applications that are real time. Simulators for off-road terrestrial applications can go beyond the dynamics capabilities of on-road simulation, while introducing higher-fidelity sensing simulation for lidar. Among these frameworks are *MAVS* [18, 19], *Chrono* [20], and *VANE* [21]. MAVS has focused on off-road sensor simulation, including modeling of lidar in vegetation and rain. Chrono use ray-tracing-based solutions for camera and lidar, allowing similar high-fidelity lidar modeling. Chrono also supports high-fidelity off-road physics, which is additionally leveraged within MAVS. While these frameworks leverage similar techniques to what is required for planetary rover and lander simulations, they do not extend to planetary simulations due to their lack of support for planetary scale rendering and lighting models.

Our *DARTS* simulation framework [22] is targeted towards engineering quality simulations of planetary exploration platform. It has included perceptions sensor models and visualization based on the *Ogre3D* raster-based graphics engine for closed-loop simulation of space and robotics autonomous platforms including planetary rovers [23]. These camera models include various lens distortions and other non-idealities [2]. The IRIS development described in this paper is a next-generation capability to provide a much higher-fidelity perception capability for the DARTS family of autonomous system simulators.

To describe the architecture of IRIS and illustrate the contributions of this work, this manuscript is organized into two parts: a description of IRIS's underlying architecture, and an overview of IRIS's specific capabilities. The general and extensible architecture is discussed in Section II, wherein the abstraction of the rendering pipeline is highlighted. Section III then introduces the rendering pipeline within the overall simulation architecture, and explains specific capabilities related to sensors, material lighting, and planetary rendering applications.

II. Rendering Framework

IRIS is built upon a general rendering and simulation framework such that any sensors, materials, data non-idealities can be introduced into the framework without modification to the underlying architecture. To this end, IRIS leverages OptiX [5] for hardware-accelerated real-time ray tracing. Ray tracing allows for ray-based modeling of sensor data acquisition, making modeling of camera, lidar, radar, etc natural in its simulation. The choice of OptiX rather than graphics APIs like OpenGL or Vulkan, or game engines like Unreal or Unity enables the abstraction of ray tracing as a compute capability, rather than a graphics capability. Specifically, this enables headless rendering with no need for a graphics context (X context), allowing operation on headless servers and super computers with no special environment configuration. Furthermore, it enables direct interoperability with CUDA, allowing for graphics-peripheral computation such as geometry manipulation and image processing, to be run directly on the GPU using the same memory as the rendering.

With the selection of OptiX as the ray tracing engine, there are a few architectural requirements necessary to interface with the CUDA-based API. First, there are *raygen*, *closest hit*, and *miss* functions which implement logic on a per-ray basis (described in more detail in Section II.B). These are the main interface mechanism with the ray tracing backend. Further, convenience features such as *direct callables* allow users to selectively determine material or lighting callbacks at runtime. IRIS interfaces with this library and provides it's own API to create complex rendered scenarios.

To leverage the capabilities of OptiX and prioritize the generality of its implementation, IRIS uses a two-level structure of abstraction. The high-level pertains to the scene modifications and data augmentations, and uses an action and filter graph to keep these components general and extensible. This is discussed in the following section. The low level pertains to the use of rays to model sensor data acquisition and materal modeling. The implemented general architecture for ray modeling is discussed in Section II.B.

A. High-level Processing Architecture

The rendering architecture described herein is structured to provide scalability, where the scene, sensors, rendering fidelity and post-processing effects are all configurable. To provide a robust and usable mechanism for modeling a wide variety of sensors with unknown attributes in various environments, it's desired that the created framework and API are both generic and expandable. IRIS utilizes building blocks that, when used together, can model a wide range of sensor characteristics. These building blocks are grouped into two paradigms: actions and filters.

In IRIS, sensors are independent of each other; when performing the rendering, different sensors will not interact with each other. As a result, each sensor has its own rendering pipeline. A sensor's rendering pipeline is the sequential logic performed by IRIS when a render is requested and when data is returned to the application. This pipeline is represented as a graph and is made up of generic actions (scene augmentations) and filters (data augmentations). An example set of actions and filters are illustrated in Fig. 3 for a camera.

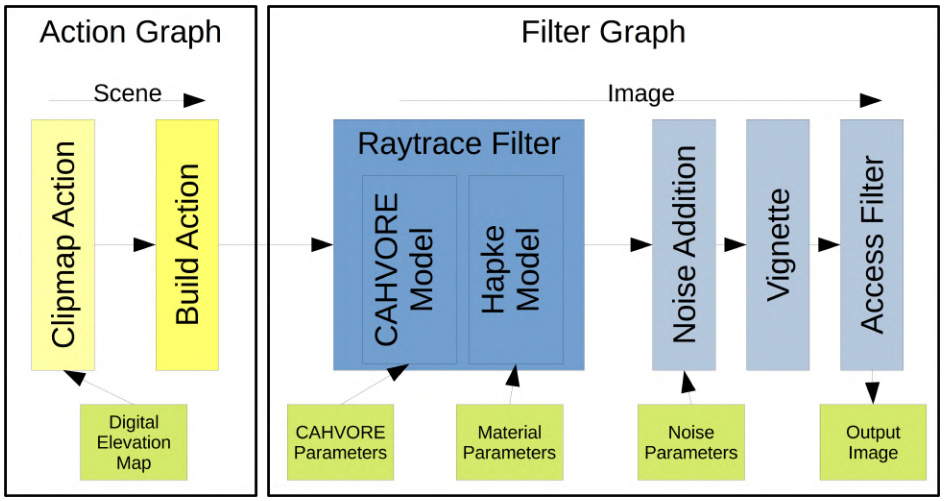


Fig. 3 Illustration of the action and filter graph for a camera in IRIS. The actions and filters are customizable to model scene modifications and data augmentations on a per-sensor basis. The ray tracing step is implemented as a filter, and serves are the data generation point for the rest of the filter graph.

1. Actions

Actions encompass generic logic performed on each sensor update prior to ray tracing. While these updates can include any change, they typically relate to changes or updates of the scene. For instance, if objects in the simulation are dynamic, the underlying physics engine may have changed the object's position, orientation, or shape since the previous render. Additionally, a sensor may need to make sensor-specific changes to the environment before rendering, e.g. geometry refinement for visual quality, parameter updates for lighting or visibility masks for debugging. Actions can perform arbitrary logic, but will not have access to the rendered data as these are performed pre-render.

The last action in the graph is a special build action which performs a scene update build in OptiX, generating a scene-level acceleration structure based on a bounding volume hierarchy (BVH). This will be used by the subsequent ray trace to optimize traversal time. An action used for geometry level of detail is discussed in Section III.D. The action graph is constructed and populated with default actions, such as the previously discussed build action; however, the graph is configurable and may be ordered as the user pleases. Further, custom actions can be implemented and added to the graph. Actions do not take in or return data, as there is no requirement of ordering (except for the build action), so all data must be passed into the action when it is constructed.

2. Filters

Similar to actions, *filters* implement generic logic that is called every time a sensor update occurs. As opposed to actions, filters receive the data buffer constructed during the ray trace step and return the buffer. Filters can be thought of as a post processing step, where sensor data augmentations can be implemented; augmentations may include image grayscaling, vignetting or simple noise modeling, or lidar filtering and noise additions.

Throughout the filter graph, data remains on the GPU. Filters can then either implement their logic on the GPU, or copy the data to the CPU to perform their augmentations, so long as the data ends on the GPU. Since data augmentations are order-dependent, each subsequent filter must be compatible with the data format of the preceding filter. This is checked at run-time as the filter graph is dynamic and can be reconfigured during the simulation.

The *render step* is implemented as a special filter that is computed first in the graph, directly following the build action. This is where OptiX is leveraged to trace rays that model the sensor's light acquisition from the scene geometry. Depending on intrinsic sensor characteristics, like field of view for a camera or beam structure for a LiDAR, rays are propagated into the scene, and fill a sensor buffer with the traced light (i.e. color for camera). This sensor data buffer serves as the buffer on which the rest of the filter graph will be computed, whether to augment this data, or visualize it. The ray trace step including ray traversal, material shading functions, and reflections, is described in detail in the following section (Section II.B).

Analogous to the action graph, the *filter graph* organizes sequential filters and directs inputs and outputs. A sensor will, therefore, have two graphs: an action graph and a filter graph. These graphs are operated on sequentially (actions, then filters), to generate the data for a given sensor. While the action and filter graphs can be extended, they will each have at least one action or filter respectively (build action and render filter). The user is free to implement and add additional actions and filters as they require. The generic nature of both actions and filters allow modularity across projects. This is advantageous considering the multitude of use cases IRIS is expected to be applied for. Throughout the simulation, filters and actions may be added, removed and reordered dynamically, as well. Further, user data access can occur at different points within the graph; for instance, a filter graph may be constructed with a grayscale filter and then a vignette filter, the user can add an access filter after the grayscale *and* after the vignette filter to grab both data buffers, one with vignetting and one without. The configurability that filters provide is powerful and is utilized to create complex sensor models.

3. Visualization

A visualization component is available in IRIS that allows for interactive viewing of the sensor output. Built on top of the open source, graphical user interface library ImGui [24], the viewport allows for mouse and keyboard input, resizing, text display, and more. Further, the visualization mechanism is versatile and can be used to view camera, lidar and other sensors. User input can be done in real time, where the user can pan, rotate and zoom in the viewing window. When a viewport window is desired for a specific sensor, a filter is added to that sensors filter graph that copies that data buffer from the GPU to the CPU at that point. When user input is made within the viewport, a re-render is requested and the action graph and filter graph are run. Multiple windows can be used at the same time for multiple sensors.

4. Multi-threading

Multi-threading and parallel computing are heavily utilized in the IRIS framework. The physics engine that IRIS collaborates with (described in Section III.A) only performs updates when a dynamics step is required. Say a rover is moving toward a goal, once that goal is reached, the physics engine will stop updating as there is no work to be done. In the case when the engine stops updating, interacting with the simulation environment is still desired, such as panning and zooming around the world. The visualization updates must be then requested on a separate thread from the physics engine since the main thread has effectively stalled.

IRIS uses threads in the following way: one for the physics engine, one for the main IRIS computing queue, one for the action and filter graphs and one for each visualization. The main IRIS computing queue is implemented such that requests can be made both from the physics engine (i.e. geometry updates) or visualizations. The computing queue will then continue to do work until no work is required. On each update, the entire filter/action graph is run.

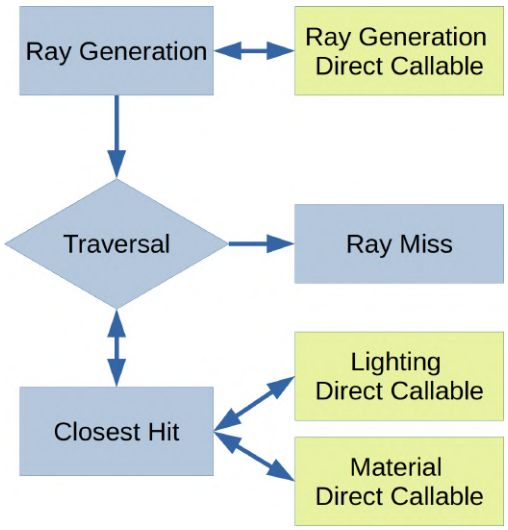


Fig. 4 Flow of the ray tracing algorithm.

B. Ray Tracing Architecture

Inside the ray tracing step, a decision was made to keep everything general such that any sensor, any material, and any light types could be seamlessly introduced in the future. This is enabled by *direct callable functions*, which are function-table abstractions for ray tracing functions, as well as general data packets carried along the rays. The *ray generation*, *closest hit*, and *miss calls* (which are defined below) make no assumptions about the ray data packets. Instead, the functions simply make calls to a function table, and then make the ray calls.

The flow of data and the definitions of the components in the algorithm is illustrated in Fig. 4. Blue boxes are the general structure that follows OptiX, green boxes are any function that can be called per object or per sensor and can implement sensor, material, or light specific functions as described below:

Ray generation: the function that is launched from the CPU once for each entry in an array corresponding to the data structure that will hold the sensor's data. For a camera, the raygen function is called for each pixel in an image.

Traversal: the algorithm that checks ray-object intersections to find the nearest collision point, if any, to the ray origin along the ray direction.

Closest hit: the function that is attached to each object and called for the object which caused the nearest hit to the ray origin along the ray direction.

Ray miss: a function that is called when no objects cause collisions along the ray direction.

Direct callable: abstract function that is implemented using a table, and called based on its index. The index can be determined from dynamic variables - fills the role of dynamically-determined template functions within the ray tracing architecture.

1. Ray Generation

The entry function is a kernel launch from within OptiX which calls the ray generation function. This is called once for each *pixel* in the array of entries (image for camera) to be traced. The ray generation function does not need to know about the sensor type or the function by which ray directions and data is determined. Instead, the ray generation function makes a call to the direct callable function table. That function calculates ray and sensor-specific information such as ray direction, and datatype (color, depth, etc). The trace calls are then made with this packed data. For ray generation, IRIS currently supports camera (pinhole, spherical, and CAHVORE), and lidar (grid-based or custom). These models are further discussed in Sections III.B.1 and III.B.2.

2. Lighting Functions

When an object is hit by the ray traversal, the closest-hit function is called. The closest-hit function is equivalent to an object-specific GPU callback function that is be triggered when the object is the source of the nearest collision for a ray along its path. Just like the ray generation function, this function also makes no assumptions about the specific sensor, materials, or lights that are in the scene. The closest hit checks the object geometry to calculate hit position and surface normal, then sends a shadow ray to each potentially visible light in the scene. Any light behind the surface is ignored for efficiency. Tracing rays to each light allows the algorithm to directly calculate shadows in a physically realistic way, and removes any need to careful shadow algorithms that are traditionally required for raster graphics. When a light is traced in IRIS, the intensity is calculated based on a direct callable. In this way, the lights are stored and calculate their own values so that the closest-hit can be agnostic of light calculations. Because of this, a new light type could be added without making any changes to the traversal algorithm. Current support includes point lights, spot lights, and directional lights.

3. Material Modeling

The same paradigm is used for material shading calculations. Once the lights are traversed in the closest-hit function, the light values are packed and sent to the material function, which is also implemented as a direct callable. The specific material function that is called make calculations to pack the ray data structure. For example, a BRDF camera material would be called when a camera hits a BRDF object. That function would then calculate the surface color, reflection, and refraction amounts and pack the ray data with these values. The color is then returned into the image, and the reflection and refraction amounts are used to traverse additional rays when requested. These material direct callables are also used for other sensors like lidar. In this case, the function packs in an intensity and direction rather than color. In exactly the same way as lights, this allows multiple material types to be used in the same simulation, and allows new materials functions to be added without reworking the render framework, even when the materials are for new sensor

types. Current material support extends to Phong, principled BRDF, and Hapke materials, which are discussed in detail in Section III.C.

III. Integration within Closed-Loop Planetary Simulations

Having described the design and architecture of IRIS, we turn to its integration and use within closed-loop dynamics simulations based on the DARTS/Dshell simulation framework for the real-time, closed-loop simulation of aerospace and robotics platforms [25]. This framework has been used for several aerospace and robotics missions and platforms including Mars 2020, Ingenuity Mars Helicopter, InSight mission, Phoenix mission, as well as technology development and demonstration efforts involving multi-agent autonomy, legged and wheeled robotic platforms, Venus aerobots, autonomous sampling, etc.

A. DARTS/Dshell simulation framework

We begin with an overview of the key middleware elements of the C++ based DARTS/Dshell framework to support sensor simulations, before turning to specific aspects of IRIS to meet the needs described in Section I.

DARTS multibody dynamics: At the heart of the simulation framework is the DARTS dynamics solver [26] for rigid and flexible multibody dynamics, where general conventions can be seen in Fig. 5. DARTS uses minimal coordinate dynamics formulations to allow the use of ODE techniques instead of the expensive DAE method that are common in the community. A further contributor to DARTS' speed is that it uses structure-based dynamics algorithms from the Spatial Operator Algebra dynamics methodology [27] that provides optimal, low-cost recursive methods for solving the rigid/flexible multibody equations of motion. The methods allow for modeling general purpose systems with tree and graph topologies, as well as with non-smooth contact and collision dynamics. The structure based property of the underlying algorithms allows the easy accommodation of configuration changes to the underlying system.

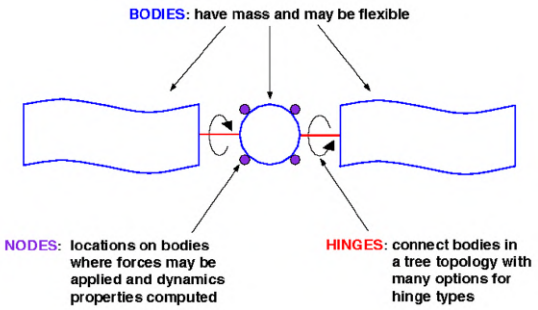


Fig. 5 The DARTS rigid and flexible multibody dynamics engine

Dshell framework: We describe here the key architectural features of Dshell that enable its multi-mission use. While Dshell uses C++ for fast performance speed, an extensive Python front-end provides analysts and developers with a convenient and sophisticated command line and scripting environment. The organizing structure for simulations within Dshell is a hierarchy of object-oriented and configurable assemblies, as seen in Fig. 6. Assemblies represent functional subsystems which can be contained within other assemblies, and can contain sub-assemblies of their own. For instance, a spacecraft assembly may contain assemblies for sensors, actuators, fuel manifolds, and these in turn can contain sub-assemblies for thrusters, valves, etc. The assemblies are configurable and parameterized to support a variety of configurations and parameter needs. Individual assemblies can be unit tested independently before use within simulations. Assemblies are responsible for the creation of component models, bodies, nodes and frames and setting up the data-flow between models. Dshell includes a library of reusable and parameterized C++ component models, along with standard simulation capabilities such as numerical integrators, events handling, finite state machines, Monte Carlo support, data logging and introspection

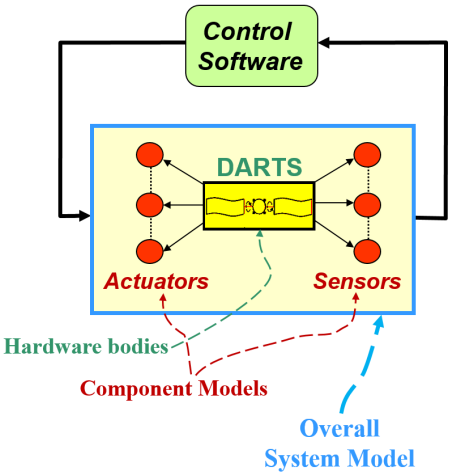


Fig. 6 The component-based Dshell simulation framework for closed-loop simulations.

capabilities.

Scene geometry layer: There is often a need to process a simulation's geometry and kinematics for a variety of purposes. For example, we may need to feed the geometry into a 3D renderer while also evaluating the same geometry for collisions. The DScene geometry layer provides an abstract interface to support a variety of such consumers of geometry, also referred to as client scenes, as seen in Fig. 7. Any client scene that implements the DScene API can be used with the simulation. This design pattern allows us to easily add new client scenes to the simulation as the need arises. During run-time, the user simply needs to register a client scene with the simulation, and it will be notified of any changes in the geometry as the simulation state evolves. The IRIS renderer includes such a DScene client scene API and so it can easily be plugged into the full closed loop simulation.

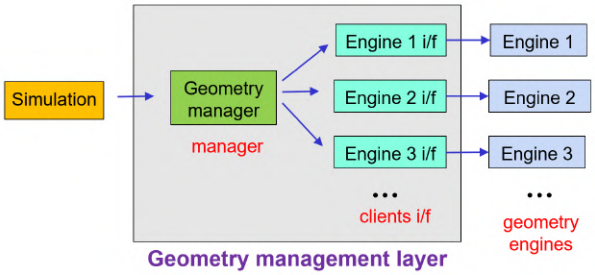


Fig. 7 The DScene geometry layer serves as an abstraction layer for interfacing the simulation with multiple geometry engines.

SimScape terrain middleware: Dshell includes a terrain modeling middleware called SimScape (Fig. 8) to model planetary bodies, local terrain patches [28]. The SimScape terrain environment toolkit facilitates the use of terrain models in a range of simulation applications. DEMs (digital elevation maps) and irregular shapes(such as asteroids) using meshes. SimScape has two primary purposes: (1) to provide tools to import terrain data from a wide range of sources into terrain models that Dshell simulations can use at run-time; and (2) to provide a library used at run-time to support loading terrains and execute queries for topography of terrains, intersection of rays with terrain surfaces, slopes, visualization, and more. SimScape is often used in Dshell simulations to model spacecraft sensors such as LIDARs, cameras, altimeters and proximity sensors as well as for landing and terrain interaction models. SimScape stores its terrain data using the Hierarchical Data Format (HDF5) data storage format for efficient and flexible access. The HDF5 data format is hierarchical and allows SimScape to deal with large data sets since selected data slices can be loaded efficiently on demand at run-time. HDF5 supports "striding", accessing every nth row or column of data to sub-sample the data.

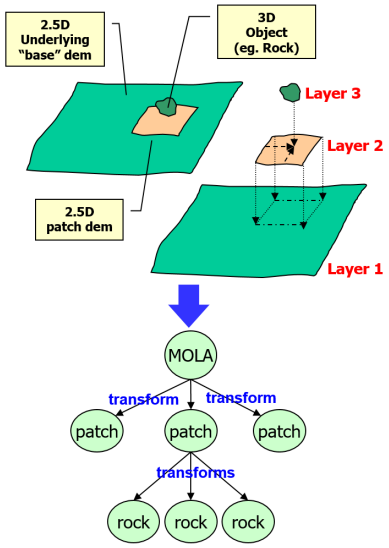


Fig. 8 The SimScape middleware for assembling terrain environment models for run-time use within Dshell simulations.

B. Perception sensors

IRIS can generate high-fidelity simulated images and data for cameras and LIDARs in real-time. Both LIDAR and camera models use direct callables, described in II.B. Each sensor defines its ray generation functions and data structures. In the case of cameras, the output data will be a 2D array of colors, and for lidars it may be a 2D array of intensity and position values. Additionally, the ray generation function specifies what data will be packed onto and carried by a ray. For cameras, this is a position, direction, accumulating color, and contribution amount so that reflections reduce the contribution of upstream light. Then, each material must know how to contribute to the specific sensor's data. This is implemented with the direct callables as discussed in Section II.B, but requires no change to the architecture itself. The is true for expanding IRIS to additional sensors like radar, or IR cameras. These would only require new definitions of material-sensor interaction functions, and new ray generation functions to specify how the rays will model the sensor's acquisition of light. The following subsections will discuss the implemented functions for cameras and lidars, in detail.

1. Camera models

IRIS supports several camera models, each described below. Each lens model is a implemented in the ray generation function, as discussed in Sec. II.B.1. Fig. 9 shows the same scene having been rendered using the three different lens models, where the camera position and orientation are kept constant.

Pinhole: The pinhole perspective projection simulates a camera with a pinhole aperture, and it is used mainly for artistic reasons and rasterization graphics with a small field of view [29].

Spherical lens: The spherical camera uses a hemisphere instead of an imaging plane, whose radius is the focal length of the camera [30].

Fish-eye lens: The fish-eye model is the projection of a sphere onto a tangent plane, and instead of equalizing the distance between pixels, as the pinhole does, it equalizes the angular distance [29]. Both projections are parameterized on the field of view.

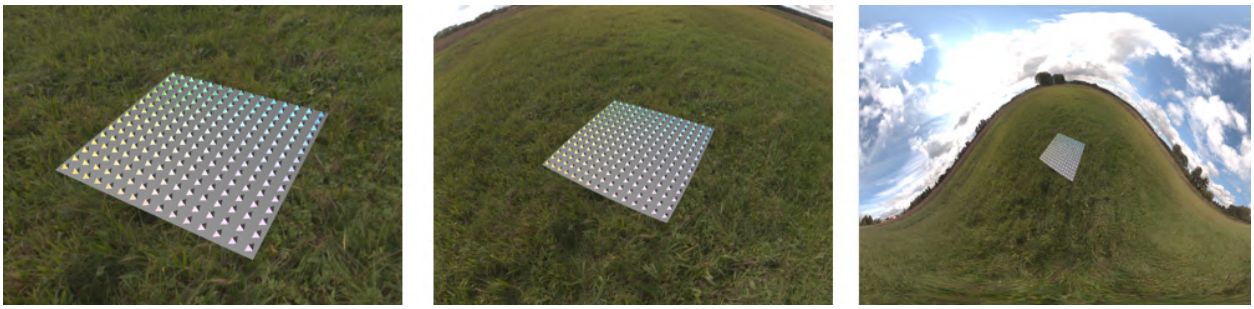


Fig. 9 From left to right: pinhole, fish-eye, and spherical camera models.

CAHV(OR(E)) camera models

CAHVOR is a camera model used in machine vision for three-dimensional measurements. It models the transformation from the object domain to the image domain using vectors C, A, H, and V and corrects radial lens distortions with a vector O and a triplet R [31]. It is a non-central camera model. The projection rays do not pass through the origin of the camera Cartesian coordinate system, but through another point C whose position depends on the angle between the ray and the camera optical axis [31].

Each letter represents a set of parameters:

- *C*: the camera center vector from the origin of the ground coordinate system (or the object to which the camera is attached, X-Y-Z) to the camera perspective center.
- *A*: the camera axis unit vector, which is perpendicular to the image plane.
- *H*: the horizontal information vector, pointing roughly rightward in the image.
- *V*: the vertical information vector, a vector pointing roughly downward in the image.
- *O*: the optical axis unit vector, which is solely used for lens-distortion correction.
- *R*: contains radial lens distortion coefficients defined in the ground (object) coordinate system (3 coefficients).
- *E*: the entrance vector to have a non-central camera model (fish-eye distortion).

The renderings in Fig. 10 are an example of CAHVORE models, starting from the simplest one, CAHV, on the right, to the most complete, CAHVORE, on the right. The parameters used for the three images are reported in Tab. 1. It can be noticed that the CAHV camera model uses the pinhole perspective projection.

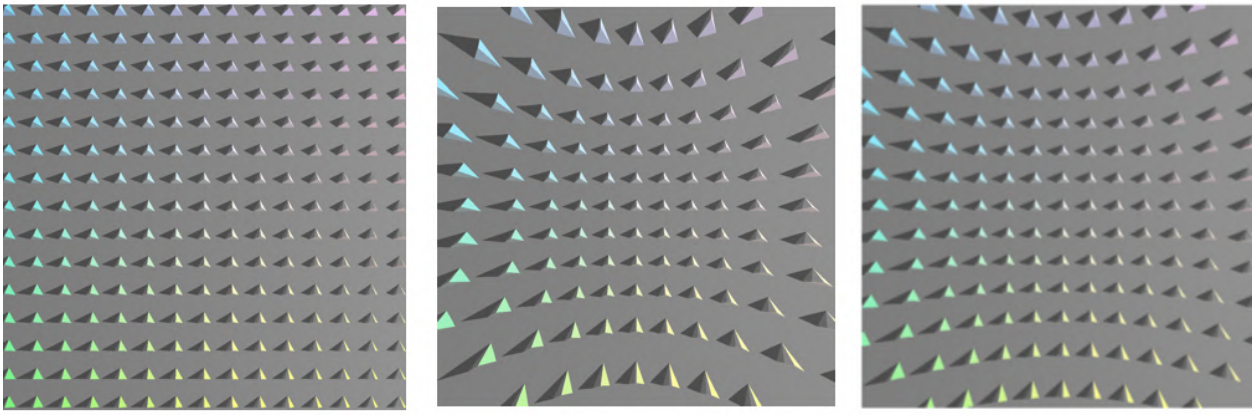


Fig. 10 From left to right: CAHV, CAHVOR, and CAHVORE camera models.

Table 1 CAHV(OR(E)) camera parameters used in Fig. 10

	C	A	H	V	O	R	E
CAHV	(0,0,35)	(0,0,1)	(512,0,512)	(0,-512,512)	-	-	-
CAHVOR	(0,0,35)	(0,0,1)	(512,0,512)	(0,-512,512)	(0,0,1)	(0,0,1)	-
CAHVORE	(0,0,35)	(0,0,1)	(512,0,512)	(0,-512,512)	(0,0,1)	(0,0,1)	(0.5,0.5,0.5)

As already stated in Section II.A.2, different types of filters can be applied to each camera regardless of its lens type. Some examples of the filters implemented in IRIS so far are shown in Fig. 11. More than one filter can be applied to a camera, as displayed in the bottom left picture of Fig. 11, where the gray scale and the Gaussian blur filters have been used.

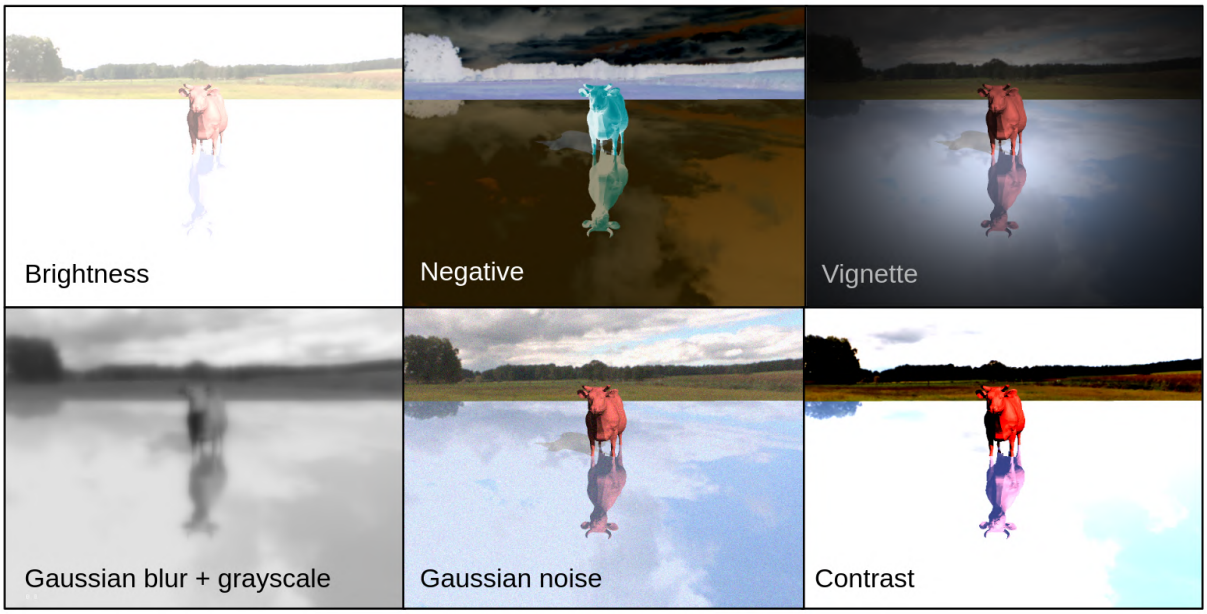


Fig. 11 Examples of different filters applied to the same scene. From left to right: *(top)* brightness, negative, and vignette filters; *(bottom)* Gaussian blur and gray scale, Gaussian noise, and contrast filters.

2. LIDAR models

A LIDAR is a sensor that emits lasers to construct a 3-D representation of the area. In robotics, LIDAR are commonly used for perception, either alone or together with cameras. Applications include surface mapping and site selection during EDL (entry, descent, and landing) and navigation of autonomous ground vehicles. LIDAR are currently in use by NASA's Ingenuity helicopter to aid in performing autonomous maneuvers on the surface of Mars.

In addition to cameras, IRIS is also capable of modeling LIDAR. Real LIDAR have a wide variety of sensor head layouts and may even rotate, so IRIS is designed with enough flexibility in LIDAR configuration to meet these disparate modeling needs. For the LIDAR raygen function we support:

- A uniform grid of beams, specified based on field of view and resolution
- A user-defined array of beams, each with an arbitrary source and direction in the sensor frame

The *uniform grid* LIDAR is useful for modeling a LIDAR with little work for commonly used LIDAR systems. The *user-defined* LIDAR offers full flexibility for the user to specify at run-time whatever odd LIDAR configuration needs to be modeled. The LIDAR can also easily be attached to a rotating frame in order to model a LIDAR rotating sensor head.

During ray tracing, each LIDAR beam is modeled using a single ray, and the output is the distance and position of the first hit for each ray. In the future we would like to expand this feature to account for reflective surfaces, partial and multiple hits, and beam divergence. IRIS also provides a way to visualize in real-time the output of a LIDAR. An on-screen OpenGL visualization of the LIDAR's hit points can be launched. The visualization shows the LIDAR hit points from an adjustable third person perspective. The points may be colored according to useful metrics such as height, distance from the LIDAR, or index in the LIDAR's beam array.

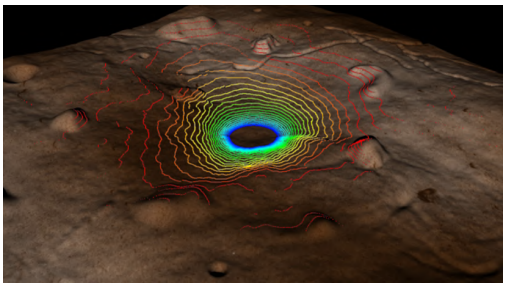


Fig. 12 An example of the OpenGL LIDAR visualization. The LIDAR hit points are rendered in a second pass on top of a corresponding camera render.

C. Material lighting models

IRIS supports multiple material types to support different lighting models such as the Hapke lighting model (described in more detail in Section III.C.2), the Principled BRDF (Bi-directional Reflectance Distribution Function), and the Phong reflection model. Hapke and Principled BRDFs fall in the definition of PBR (Physically Based Rendering) materials since they describe the visual properties of a surface in a physically accurate way.

The Principled BRDF is a general-purpose physics-based shader created by Disney [32], and it is capable of accurately describing how light interacts with a wider variety of materials, such as metals, dielectrics, plastics, and paints. The Phong reflection model is not a PBR material, but it has been an industry standard for over thirty years, mainly used for plastics. In Fig. 2, all of these three models have been used: Hapke for the terrain, Principled for Ingenuity, and Phong for Perseverance.

Radiative transfer is the physical phenomenon responsible for the energy transfer when the light interacts with matter. Therefore, being able to reproduce this model is crucial for space simulations and computer vision. Various numerical methods for the solution of the radiative transfer equation for light scattering have been developed to deal with planetary surfaces. However, most of these models are empirical and represent an approximation with no attempt to represent the scattering behavior physically, [33]. We describe next the various lighting models supported within IRIS.

1. Other light scattering models

Lumme and Bowell developed independent models to deal more realistically with regoliths, including effects of multiple scattering, microstructure, and surface roughness, [34]. The most significant difference from Hapke is that the roughness affects only the single-scattering term.

One of the most simplified scattering models is the Lambert reflection law, a multiple-scattering law that describes perfectly diffuse surfaces. This law, Eq. 1, is more suitable for high-albedo surfaces such as snow, [33], and is defined by

$$r(i, e) = \frac{1}{\pi} \cdot A_L \cdot \mu_0 \, , \tag{1}$$

where A_L, called Lambert's albedo, is 1 for a perfectly diffusive surface.

The Minnaert reflection law is a generalization of Lambert's law. As shown in Eq. 2, it depends on both the incident angle and emission angle, since μ_0 and μ are the cosines of these two angles, respectively. This law describes the reflectance well at small phase angles α, [35].

$$r(i, e) = A_M \cdot \mu_0^k \cdot \mu^{k-1} \,, \tag{2}$$

where A_M is named Minnaert albedo and k is the limb darkening parameter.

Another model which is suitable for uniform particles in low-albedo surfaces is based on the Lommel-Seeliger law shown in Eq. 3. It is a single-scattering model, and it can be considered a simplified version of the Hapke model, [33].

$$r(i, e, \alpha) = \frac{w}{4\pi} \frac{\mu_0}{\mu_0 + \mu} \tag{3}$$

2. Hapke lighting model

Bruce Hapke presented a model to deal with particulate surfaces more realistically. He was able to include the effects of microstructure, opposition effects, anisotropic single-scattering, isotropic multiple scattering, and macroscopic roughness. For these reasons, this lighting model is more accurate than the previous ones.

The radiance coefficient $r(i, e, \alpha)$ describes the relation between the reflected radiance I in the camera direction to the incident solar irradiance F, as a function of incident angle i, emission angle e, and phase angle α, as shown in Fig.13, [33].

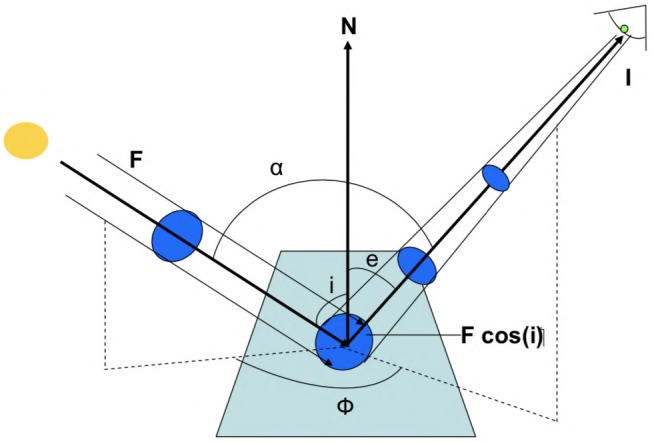


Fig. 13 Light's interaction with the surface: α is the phase angle, e and i are the emission and the incident angles, ϕ is the azimuth angle.

The azimuth angle, defined as ϕ, is the angle between the projections of the radiance and irradiance. It lies within the interval $[0, \pi]$. The bidirectional reflectance equation, proposed in [36], takes into account for all the effects cited earlier, as shown in Eq.4.

$$r(i, e, \alpha) = \frac{w}{4\pi} \frac{\mu_{0e}}{\mu_{0e} + \mu_e} \left([1 + B_{sh}(\alpha)]\, p(\alpha) + H(\mu_{0e}) H(\mu_e) - 1 \right) (1 + B_{cb}(\alpha))\, S(i, e, \phi) \,, \tag{4}$$

where:

- $\frac{1}{4\pi} \cdot w \cdot \frac{\mu_0}{\mu_0 + \mu}$ represents the reflectance per solid angle due to single scattering, [37].
- The function $p(\alpha)$, called single-particle phase function, describes the angular pattern of scattering for irregular particles. It is generally modeled either with a single, double, or triple term empirical Henyey-Greenstein phase function, [38].

- $H(\mu_0)$ and $H(\mu)$ are the analytical approximations to the Chandrasekhar's H function for isotropic scatterers, which do not have a closed-form, [38].
- $B_{sh}(\alpha)$ and $B_{cb}(\alpha)$ are the shadow-hiding and coherent backscatter opposition effect, respectively. The opposition effect is a phenomenon that causes the inability to observe terrain detail, in all porous particulate media, at low phase angles α (at opposition), [39].
- $S(i, e, \phi)$ is the general macroscopic roughness, [36]. The surface is assumed to be made up of facets tilted at different angles, which are distributed uniformly in azimuth, as shown in Fig.14.

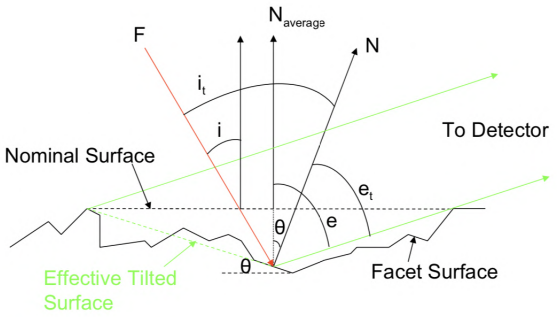


Fig. 14 Surface roughness.

The Hapke model uses a set of parameters to empirically describe the directional reflectance properties of various regolith bodies in the solar system. These are called Hapke parameters, and they are the single scattering albedo, the width and the strength of the shadow-hiding and coherent backscatter opposition effects, the asymmetry factor used in the phase function, and the macroscopic roughness angle, also named effective surface tilt. The single scattering albedo is the ratio of scattering efficiency to total light extinction, which includes scattering and absorption.

Currently, the values used for the Hapke parameters are just constants, but textures could be supported to have variability, as we are already doing for the albedo and normal maps. In Fig. 15, the Hapke lighting model has been applied to the asteroid Itokawa at different opposition angles α. The Hapke parameters used for Itokawa are reported in Tab. 2, [33]. In the rendering at the center, the brighter spot due to the opposition effects is visible in the middle, causing a loss of details around it. Here the α angle is close to zero, while in the rendering on the right and left, the sun and the camera direction are moving apart. The anisotropic single-scattering term $\frac{w}{4\pi}\frac{\mu_0}{\mu_0+\mu}$ can be noticed in the rendering at the extremes, and it is responsible for the increase of brightness close to the limb at low phase angles.

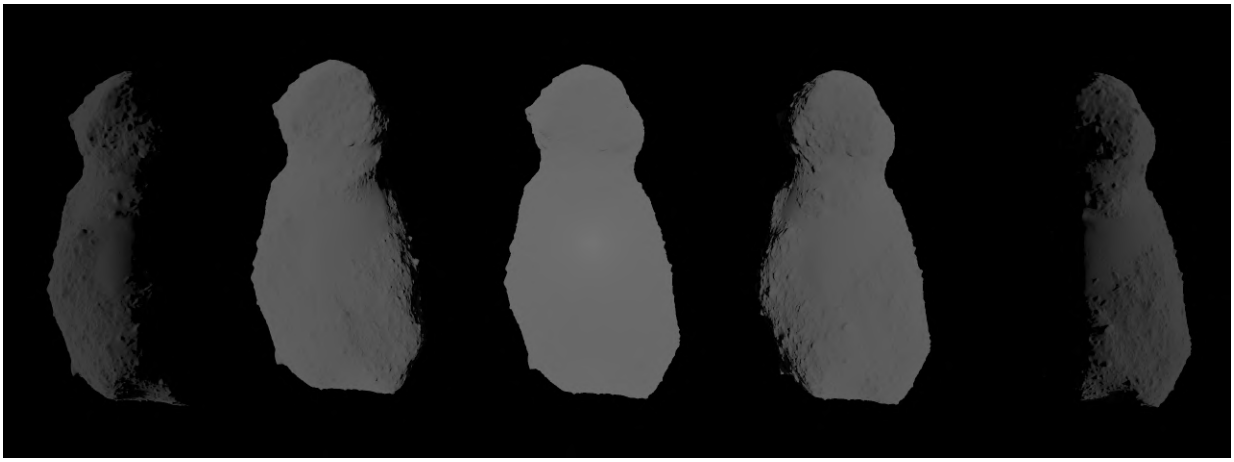


Fig. 15 Hapke lighting model applied to Itokawa asteroid, at different opposition angles.

Table 2 Hapke parameters for asteroid Itokawa, Fig. 15

Asteroid	w	g	h_{SH}	$B_{0,SH}$	h_{CB}	$B_{0,CB}$	θ
Itokawa	0.42	-0.35	0.01	0.87	-	-	26

D. Geometric Modeling from Large Planetary Data Sets

A critical aspect of simulating planetary missions is reconstruction of an environment that is as close as possible to the mission destination. To accommodate this, large terrain and planetary data sets can be leveraged to reconstruct surface geometry. These data sets are typically comprised of digital elevation maps (DEM) that store the elevation of a terrain in an image using pixel values to define precision in height. Since these data sets can be massive, upwards of 100 million pixels, it is infeasible to render the entire data set. In addition, the full resolution of the terrain may not be necessary. Consider a planetary data set. For a mission far from the surface, the planet could be rendered using a low resolution version of the terrain. For a mission on a planet's surface, higher resolution would be needed, but a much smaller patch of terrain would be visible and therefore required for rendering.

To handle this problem dynamically, IRIS uses geometry clipmaps, as discussed in [40, 41]. This rendering technique has been used in graphics for some time, but combining this technique with the available planetary data sets for simulation, allows for complex simulations of orbiters, landers, and rovers to make sure of complex detailed terrain while maintaining interactive ray tracing frame rates. The clipmap is implemented in an action in the action graph, and thus can make arbitrary changes to the scene before rendering. The action is attached to a sensor, and uses the sensor location to find a focal point where the terrain should be at it's highest resolution. Then, optionally using a separate worker thread, generates a geometric representation of the data set described in a terrain height data set.

To optimize performance, data sets are stored in HDF5 format, such that the DEM data can be read in directly using multiple strides. The data that makes up the clipmap uses discrete levels of the same side dimension ($2^k - 1$ triangles). The first and finest resolution is placed at the region of interest using the side dimension specified. Each subsequent level with use a triangle size twice that of it's interior level, resulting in a terrain that decreases in resolution the further it is away from the region of interest. As theregion of interest moves, so does the focus of the clipmap. Care is taken to only update the focal position and levels as needed for the update.

When updating the clipmap, data is read from the file in patches to reduce the total file read time. Once the DEM patch is loaded into memory, it is queried to find the terrain height and vertex normal to ensure smooth shading across the terrain. Additionally, texture coordinate information must be calculated and added to the vertices such that the intended terrain texture can be stretched across the clipmap. Finally, the vertices must be arranged into triangles, and the seams between levels must be stitched together. The performance of a CPU-based implementation and GPU-based implementation are show in Fig. 16. While the CPU-based implementation requires no additional GPU memory, the GPU-based implementation is much more efficient as it queries data from the DEM in large patches, and make all per-vertex and per-triangle calculations in parallel.

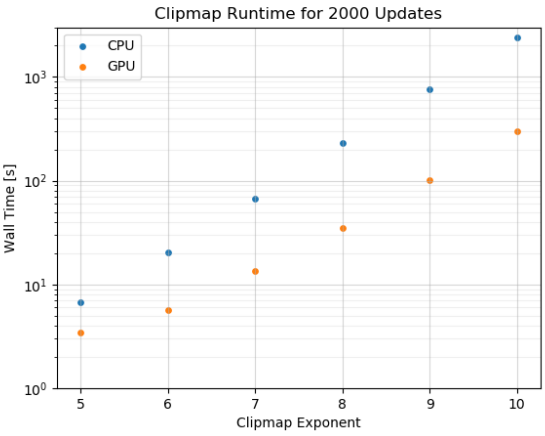
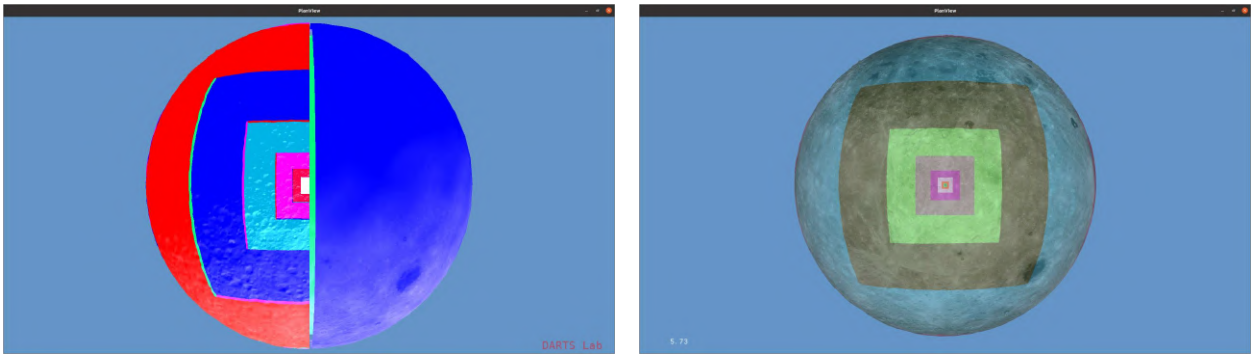


Fig. 16 Performance comparison of clipmap implementation between CPU and GPU. Results show the time required for updating the full clipmap as a function of the clipmap exponent which defines the number of triangles in each clipmap level.

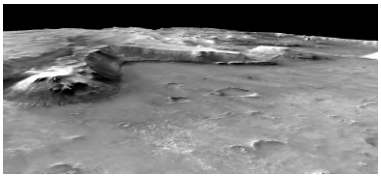
For planetary data sets, additionally mapping is done to wrap the clipmap around the data set's seam, as shown in Fig. 17. Without the wrapping, simulations near the seam could see sharp changes in terrain resolution. This is clear in Fig. 17a which colors individual levels using colors to make them visible. Instead, the levels are wrapped across the seam, resulting in a continuous level of detail, even near the data seam. This is demonstrated in Fig. 17b with the same viewing location, where the color of each level is continuous across the seam, making the seam invisible. Example renderings using the clipmap implementation are shown in Fig. 18, where scenes including the Jezero DEM, the Bennu asteroid, and the Lunar global data set are shown.



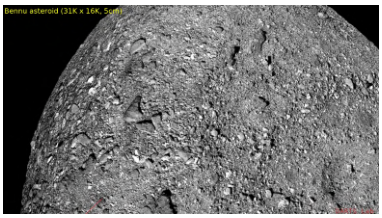
(a) Clipmap colored by layers without wrapping

(b) Clipmap colored by layers with wrapping

Fig. 17 Wrapping of clipmap around planet. Allows for missions near data seam to have consistent level of detail as if it were not near the seam.



(a) Jezero crater (100 million vertices)



(b) Bennu asteroid (500 million vertices)

(c) Lunar global data (4 billion vertices)

Fig. 18 Planet and terrain examples showing the level of detail obtainable using clipmapping while achieving interactive performance.

E. Additional Scene Rendering Features

In addition to geometric and material lighting models, IRIS provides extra visualization capabilities that enable visualizing missions and include debug information. For example, it may be important to visualize rover wheel tracks, the trail of the rover across the terrain, or the bounding boxes of specific bodies in the simulation. While some of these are not physically contained in the scene, it may provide insightful or critical information for engineering understanding of the simulation results. This section outlines these capabilities in IRIS.

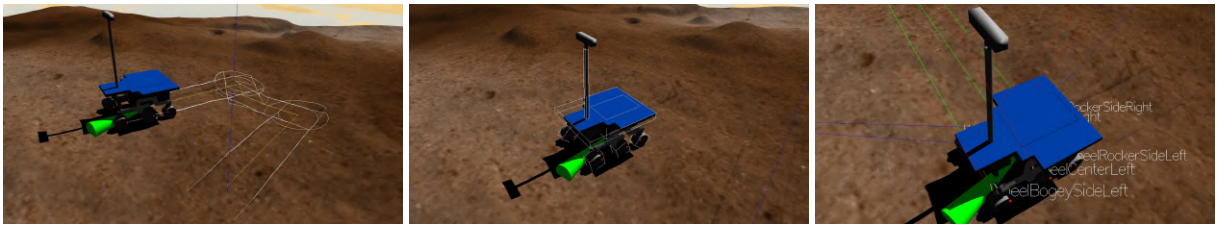
Wheel tracks. In some studies, it may be important for a rover's wheels to leave behind visual tracks that could have an impact on the perception stack. A low-cost way to achieve this effect is to perturb the terrain's normal vector when a ray intersection occurs at the location of a wheel track. In other words, the terrain's geometry is not altered, but light interacts with the terrain differently in places where wheel tracks are present, creating the illusion of altered geometry. In practice this is accomplished by maintaining a circular buffer of the N most recent track segments. As a vehicle moves, new tracks are added to the buffer and the oldest ones are discarded. The buffer is made available to the ray intersection shader in Optix, which determines if and how the terrain normal should be perturbed based on the presence of a wheel track.

Frames. For human-friendly visualization, it is frequently useful to be able to display the position and orientation of a reference frame. IRIS supports visualizing reference frames either as a 3D object, using arrows constructed from cones and cylinders or as a set of three line segments in screen-space, meaning their width in pixels can remain constant. These fixed-pixel-width lines are accomplished by implementing them in a second render pass. This second render pass leverages raster graphics via OpenGL, which is more natural when rendering with fixed-pixel widths. This second render pass can be added as a filter to specific cameras, and uses depth-testing to seamlessly add the 3D rastered information into the ray-traced image. Whether using the line-based or shape-based frames, these are excluded from shadowing to prevent lighting changes to the scene. An example of frame visualization is shown in Fig. 19c.

Trails. We often want to visualize the past trajectory of a vehicle, or more generally the trajectory of some reference frame. IRIS is capable of visualizing the history of any frame with respect to any other frame as a trail, which is represented as a sequence of line segments. Trails are rendered as fixed-pixel-width lines using the same OpenGL render pass described for frames. An example of trail visualization can be seen in Fig. 19a.

Bounding Boxes. IRIS can additionally render the edges of an object's bounding box for visualization and debugging purposes. Similar to trails, these are represented as a collection of line segments which are rendered using OpenGL. The depth testing here is critical to ensure the occluded edges are not visible. An example of bounding box visualization is shown in Fig. 19b.

Text overlays. IRIS has support for including text in the render, either as an overlay drawn at a fixed position on the screen, or as an object with a 3D location in the scene, attached to some reference frame. In the case of overlay text, the text is added to the GUI used for displaying the image. In the case of text in the 3D scene, the OpenGL render pass is leveraged to introduce the text, ensuring the text remains attached to its frame, and correctly depth tested and excluded from shadowing. An example of text rendering for frames can be seen in Fig. 19c.



(a) Trails **(b) Bounding Boxes** **(c) frames**

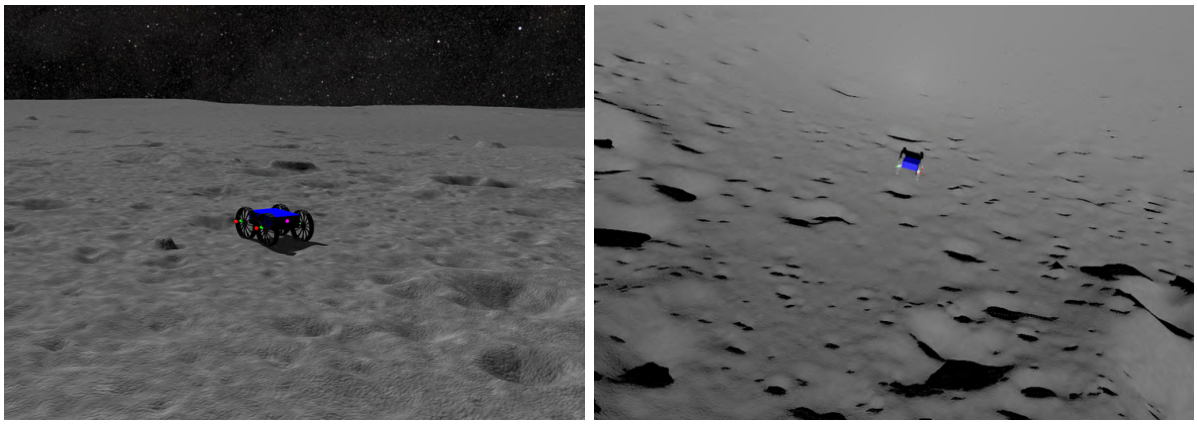
Fig. 19 Scene ornaments rendered with OpenGL second render pass.

Origin Frame. We often have a need to produce renders from scenes on the scale of the solar system. It is convenient for simulation purposes to choose the solar system barycenter as the inertial frame, but this can lead to precision related artifacts during rendering. To meet these conflicting needs, we have developed the capability to decouple the origin frame used for simulation from the origin frame used for rendering. The simulation is the source of truth for all frame transforms and does computations in double precision, which is generally sufficient for scenes on the scale of the solar system. Before publishing frame transforms to the renderer, IRIS performs the necessary computations to change the scene origin to the one required for rendering, typically at or near the sensor. This allows us to quickly render solar system scale scenes in single precision without producing artifacts.

F. Applications

The IRIS toolkit is currently being used by NASA's *Cooperative Autonomous Distributed Robotic Explorers (CADRE)* project [42]. CADRE's goal is to develop and demonstrate multi-agent autonomy technologies via multiple rovers on the Moon's surface. IRIS is used for sensor modeling and visualization within a DARTS/Dshell multi-rover simulation for the CADRE autonomy software development and test. The rover, camera sensor and environment simulations for CADRE thus include all the features described in the previous sections, such as CAHVORE camera models, various filters, large terrain clipmapping support, Phong and PBR materials, Hapke lighting model for the terrain and wheel tracks.

Figure 20a shows the Scout vehicle on the Reiner Gamma Lunar 2 cm resolution terrain. Hapke lighting model has been used for the terrain. The opposition effect is visible close to the rover in Fig. 20b. Phong lighting model has been used for the rover materials.



(a) Scout rover on lunar terrain landing site Reiner Gamma.

(b) Opposition effects visible close to Scout rover on lunar terrain landing site Reiner Gamma.

Fig. 20 Rendering of Scout rover on lunar terrain using IRIS.

IV. Conclusion and Future Work

Meeting the demanding set of sensor modeling requirements is challenging, and usually requires a significant trade-off between simulation speed and modeling fidelity. This paper describes the new IRIS sensor modeling capability for the real-time, high-fidelity simulation of vision sensors that avoids such compromises. IRIS takes advantage of GPU-accelerated Nvidia's OptiX 7 ray tracing and CUDA software to meet the fidelity and real-time performance requirements. The IRIS toolkit has been integrated within JPL's DARTS/Dshell [25] spacecraft and robotics simulator for closed-loop simulations of autonomous space vehicles, landers, small bodies, and robotics applications.

IRIS can generate accurate simulated images and data for cameras and LIDARs in real-time. The camera models include various lenses such as pinhole, fish-eye and spherical, supporting CAHVORE models. The IRIS toolkit includes post-processing filters to model sensor non-idealities such as Gaussian noise, blur, vignetting, gray-scale, and color inversion. The user can configure the filter pipeline and can change parameters at run-time.

IRIS can also render in real-time large, natural, and synthetic planetary data sets with even billions of vertices such as entire planetary bodies and high-resolution digital elevation maps (DEMs) of landing site regions. Handling large data sets in real-time is possible via a continuous level of detail clipmapping method. Unlike common level of detail methods, clipmapping does not require any pre-processing of the terrain data. The clipmapping algorithm builds on the SimScape middleware for handling planetary terrain data, which supports the importing of geo-referenced data in many formats such as GeoTiff, obj, stl, gltf, and mesh.

Furthermore, the IRIS toolkit includes the advanced Hapke illumination model, which is based on a physical description of the multiple scattering behavior of a particulate surface and accurately describes how the light interacts with a regolith body. The model includes the effects of microstructure, anisotropic single-scattering, isotropic multiple-scattering, and macroscopic roughness for regolith surfaces. The Hapke model, in particular, correctly models the opposition effect, which is the degradation in observed terrain detail for regolith surfaces at low phase angles. IRIS also supports other lighting models to describe different material types, such as the Phong reflection model and the Principled BRDF. All of these lighting models have been implemented within IRIS as CUDA shaders, and the rendering pipeline allows switching between them and adjusting their properties at run-time.

Rendering performance with IRIS depends on the complexity of the scene and the materials applied. However, for an average/complex scene, the number of frames per second is around sixty on a PC with 32 GB of RAM and an NVIDIA QUADRO RTX 5000 graphics card. This performance is usually adequate for real-time, closed-loop simulation use. The pipeline is flexible and additionally allows for a second OpenGL-based render pass that can add text, lines, and boxes into scene for visual and debugging purposes. This is also used to add rover trails, object frames, labels, and bounding boxes to the scene.

Future work includes implementing path-tracing method for shaders, along with the option to choose between path and ray tracing at run-time. Path tracing is similar to ray tracing, but instead of shooting a second ray towards the light, it casts many randomized rays from that pixel and averages the result over time. With more rays to process, the

computational cost of path tracing is higher, but this technique can capture the effect of other objects in the scene and indirect lighting. Denoiser will be an option for path tracing to get immediate noise-free visual feedback, even with more complex rendering scenarios. The denoiser takes the existing renderings and applies a denoising operation to them, averaging the multiple samples from path tracing and smoothing out areas where noise is present. There is ongoing work on IRIS to support more PBR materials, camera lens shaders. Moreover, refractive materials, such as glass, liquids, or transparent plastics, will be introduced. We are also planning to include more filters and post-processing effects, like camera dirt and atmospheric dust model, to increase the sensor fidelity.

Acknowledgments

©2021 California Institute of Technology. Government sponsorship acknowledged. This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] "Phong shading," https://en.wikipedia.org/wiki/Phong_shading, 2021.

[2] Madison, R., Pomerantz, M., and Jain, A., "Camera response simulation for planetary exploration," *European Space Agency, (Special Publication) ESA SP*, 2005.

[3] "Blender," http://www.blender.org, 2021.

[4] "povray," http://www.povray.org/download/, 2004. Computer software.

[5] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M., "OptiX: A general Purpose Ray Tracing Engine," *ACM Transactions on Graphics*, 2010.

[6] Parkes, S., Dunstan, M., Matthews, D., Martin, I., and Silva, V., "LIDAR-based GNC for Planetary Landing: Simulation with PANGU," *DASIA 2003-Data Systems In Aerospace*, Vol. 532, 2003.

[7] Martin, I., Dunstan, M., and Gestido, M. S., "Planetary Surface Image Generation for Testing Future Space Missions with PANGU," *2nd RPI Space Imaging Workshop*, 2019.

[8] Scharringhausen, M., and Witte, L., "An Efficient and Lightweight Illumination model for Planetary Bodies including Direct and Diffuse Radiation," *Journal of Computational Science*, 2019.

[9] Brochard, R., Lebreton, J., Robin, C., Kanani, K., Jonniaux, G., Masson, A., Despré, N., and Berjaoui, A., "Scientific image rendering for space scenes with the surrender software," *arXiv preprint arXiv:1810.01423*, 2018.

[10] Allan, M., Wong, U., Furlong, P. M., Rogg, A., McMichael, S., Welsh, T., Chen, I., Peters, S., Gerkey, B., Quigley, M., et al., "Planetary rover simulation for lunar exploration missions," *2019 IEEE Aerospace Conference*, IEEE, 2019, pp. 1–19.

[11] Aguero, C., Koenig, N., Chen, I., Boyer, H., Peters, S., Hsu, J., Gerkey, B., Paepcke, S., Rivero, J., Manzo, J., Krotkov, E., and Pratt, G., "Inside the Virtual Robotics Challenge: Simulating Real-Time Robotic Disaster Response," *Automation Science and Engineering, IEEE Transactions on*, Vol. 12, No. 2, 2015, pp. 494–506.

[12] Open-Source-Robotics-Foundation, "A 3D multi-robot Simulator with Dynamics," http://gazebosim.org/, ???? Accessed: 2015-03-09.

[13] NVIDIA, "NVIDIA Isaac Platform for Robotics," https://www.nvidia.com/en-us/deep-learning-ai/industries/robotics/, 2020. Accessed: 2018-07-06.

[14] Shah, S., Dey, D., Lovett, C., and Kapoor, A., "AirSim: High-fidelity visual and physical simulation for autonomous vehicles," *Field and service robotics*, Springer, 2018, pp. 621–635.

[15] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V., "CARLA: An Open Urban Driving Simulator," *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[16] NVIDIA, "NVIDIA DRIVE Constellation," https://www.nvidia.com/en-us/self-driving-cars/drive-constellation/, 2018. Accessed: 2018-02-06.

[17] CVEDIA, "Syncity," https://www.cvedia.com/syncity/, 2020. Accessed: 2020-02-06.

[18] Carruth, D. W., "Simulation for Training and Testing Intelligent Systems," *World Symposium on Digital Intelligence for Systems and Machines (DISA)*, 2018, pp. 101–106.

[19] Goodin, C., Doude, M., Hudson, C., and Carruth, D., "Enabling off-road autonomous navigation-simulation of LIDAR in dense vegetation," *Electronics*, Vol. 7, No. 9, 2018, p. 154.

[20] Elmquist, A., Serban, R., and Negrut, D., "A Sensor Simulation Framework for Training and Testing Robots and Autonomous Vehicles," *Journal of Autonomous Vehicles and Systems*, Vol. 1, No. 2, 2021, p. 021001.

[21] Goodin, C., George, T., Cummins, C., Durst, P., Gates, B., and McKinley, G., "The Virtual Autonomous Navigation Environment: High Fidelity Simulations of Sensor, Environment, and Terramechanics for Robotics," *Earth and Space*, 2012, pp. 1441–1447.

[22] Cameron, J. M., Jain, A., Dan, B., Bailey, E., Balaram, J., Bonfiglio, E., Grip, H. F., Ivanov, M., and Sklyanskiy, E., "DSENDS: Multi-mission Flight Dynamics Simulator for NASA Missions," *AIAA SPACE 2016*, Long Beach, CA, 2016.

[23] Jain, A., Guineau, J., Lim, C., Lincoln, W., Pomerantz, M., Sohl, G. t., and Steele, R., "Roams: Planetary Surface Rover Simulation Environment," *International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS 2003)*, Nara, Japan, 2003.

[24] "ImGui," `https://github.com/ocornut/imgui`, 2021.

[25] "Dynamics And Real-Time Simulation (DARTS)," `https://dartslab.jpl.nasa.gov`, 2021.

[26] Jain, A., "DARTS - Multibody Modeling, Simulation and Analysis Software," *Multibody Dynamics 2019*, edited by A. Kecskeméthy and F. Geu Flores, Springer International Publishing, Cham, 2020, pp. 433–441.

[27] Jain, A., *Robot and Multibody Dynamics: Analysis and Algorithms*, Springer, 2011.

[28] Jain, A., Cameron, J., Lim, C., and Guineau, J., "SimScape terrain modeling toolkit," *2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)*, IEEE, 2006, pp. 8–pp.

[29] Marrs, A., Shirley, P., and Wald, I. (eds.), *Ray Tracing Gems II*, Apress, 2021.

[30] Rituerto, A., Puig, L., and Guerrero, J., "Comparison of omnidirectional and conventional monocular systems for visual SLAM," *10th OMNIVIS with RSS*, 2010.

[31] Madison, R., Pomerantz, M., and Jain, A., "Camera response simulation for planetary exploration," 2005.

[32] Burley, B., and Studios, W. D. A., "Physically-based shading at disney," *ACM SIGGRAPH*, Vol. 2012, vol. 2012, 2012, pp. 1–7.

[33] Spjuth, S., "Disk-resolved photometry of small bodies," Ph.D. thesis, Technische Universität Braunschweig, 2009.

[34] Lumme, K., and Bowell, E., "Radiative transfer in the surfaces of atmosphereless bodies. I-Theory. II-Interpretation of phase curves," *The Astronomical Journal*, Vol. 86, 1981, pp. 1694–1721.

[35] Minnaert, M., "The reciprocity principle in lunar photometry," *The Astrophysical Journal*, Vol. 93, 1941, pp. 403–410.

[36] Hapke, B., "Bidirectional reflectance spectroscopy: 3. Correction for macroscopic roughness," *Icarus*, Vol. 59, No. 1, 1984, pp. 41–59.

[37] Burbine, T. H., *Asteroids*, Vol. 17, Cambridge University Press, 2016.

[38] Hapke, B., "Bidirectional reflectance spectroscopy: 5. The coherent backscatter opposition effect and anisotropic scattering," *Icarus*, Vol. 157, No. 2, 2002, pp. 523–534.

[39] Aiazzi, C., Quadrelli, M., Gaut, A., and Abhinandan, J., "Physics-based rendering of irregular planetary bodies." *i-SAIRAS*, 2020.

[40] Losasso, F., and Hoppe, H., "Geometry clipmaps: terrain rendering using nested regular grids," *ACM Siggraph 2004 Papers*, 2004, pp. 769–776.

[41] Myint, S., Jain, A., Cameron, J. M., and Lim, C., "Large terrain modeling and visualization for planets," *Proceedings - 4th IEEE International Conference on Space Mission Challenges for Information Technology, SMC-IT 2011*, 2011.

[42] "Cooperative Autonomous Distributed Robotic Explorers (CADRE)," `https://www.nasa.gov/directorates/spacetech/game_changing_development/projects/CADRE`, 2021.