AAS 23-122

IMAGE RENDERING AND TERRAIN GENERATION OF PLANETARY SURFACES USING SOURCE-AVAILABLE TOOLS

Jacopo Villa[*], Jay W. McMahon[†], Issa A. Nesnas[‡]

Simulating planetary surface imagery is critical for tasks such as mapping, optical navigation, and scientific exploration. It is also vital for training and validating autonomous surface navigation algorithms. Such applications necessitate high-fidelity synthetic images that authentically mirror the unique properties of planetary surfaces, including their reflectance, albedo, and topography. In this paper, we introduce rendering pipelines based on Blender Cycles and Unreal Engine 5, an open-source and source-available software tool, respectively. While Blender Cycles offers superior fidelity imagery via path tracing at the expense of computational costs, Unreal Engine 5 provides real-time rendering capabilities with photorealistic results. Using these tools, we delve into the integration of user-defined reflectance models, such as the Hapke model, and explore procedural-terrain-generation techniques to create entirely synthetic celestial bodies. Our demonstrations include replicating an actual image from the Rosetta mission, rendering an image of the Moon, and producing procedurally-generated asteroids using Blender. Additionally, we use Unreal Engine 5 to create a procedural rubble-pile asteroid, comprising thousands of surface rocks and amounting to billions of triangular mesh elements, all rendered in real time. Despite some constraints, we conclude that these tools show promising potential as resources for training and testing autonomous navigation algorithms.

INTRODUCTION

As space missions strive for increasingly ambitious objectives and operations, we are entering a realm of remarkable capabilities and scientific discoveries. From successful surface sampling of rubble-pile asteroids in microgravity environments by NASA's OSIRIS-REx[1] and JAXA's Hayabusa2 missions,[2] to the high-speed intercept and impact with a binary asteroid system accomplished by NASA's DART's spacecraft, the trajectory of space exploration is escalating.[3] Upcoming missions are no less audacious, such as ESA's Comet Interceptor,[4] which aims to conduct a flyby of a long-period comet, and NASA's Dragonfly[5] mission to Saturn's moon, Titan, intended to study its chemistry and potential habitability. Given the growing complexity of these missions, onboard autonomy is becoming increasingly enabling for numerous spacecraft subsystems, including Guidance, Navigation, and Control (GNC), particularly for operations where long turnaround times or human-in-the-loop interactions are infeasible or impractical.

Onboard cameras serve as vital sensors when navigating spacecraft in the vicinity of planetary bodies and conducting scientific investigations. While cameras provide valuable information about the spacecraft's surroundings, the data they capture can be complex to process and use onboard. Nevertheless, recent successes with advanced onboard, vision-based navigation algorithms during high-stakes mission operations, like OSIRIS-REx's Natural Feature Tracking during the Touch-and-Go maneuver[6] and Mars 2020's Lander Vision System[7] for Entry, Descent, and Landing, demonstrate the potential of these systems for future GNC strategies.

[*]Graduate Research Assistant, Aerospace Engineering Sciences, University of Colorado Boulder, 3775 Discovery Drive, Boulder, CO 80303

[†]Associate Professor, Aerospace Engineering Sciences, University of Colorado Boulder, 3775 Discovery Drive, Boulder, CO 80303

[‡]Principal Robotics Technologist, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109, USA

However, testing and validating vision-based navigation techniques can be challenging. Beyond modeling the camera sensor, optics, and associated error sources, difficulties arise when simulating the observed scene of a planetary body. These complications, which stem from the rugged surface topography, produce: (1) complex shadowing effects, (2) unique lighting properties of regolith terrains that require advanced reflectance models, and (3) diversity within surface features and composition that lead to local albedo variations. This is especially true for small celestial bodies or scenes observed at low altitudes where these effects become more pronounced.

Despite these challenges, high-fidelity simulation of planetary surfaces is critical not just for testing and validation purposes, but also for training algorithms that rely on large amounts of data to function properly, such as machine learning techniques. This is particularly important for deep-space mission scenarios, as we currently have a limited amount of planetary imagery that can be used for training purposes. In this context, the demand for tools that can rapidly simulate and generate artificial, yet convincingly realistic, planetary bodies and terrains is escalating.

Modern techniques to simulate imagery of complex scenes rely on Physically Based Rendering (PBR), where the behavior of the observing camera, light, and materials interacting with light is modeled based on physical principles. In the broader rendering and Computer-Generated Imagery (CGI) field, many tools are available and used across a variety of fields and industries, such as animations and video gaming. Two very common tools are Blender* and Unreal Engine†. Blender is an open-source, versatile software for 3D computer graphics that is widely used for applications ranging from rendering to 3D sculpting. For our purposes, it is particularly relevant for its accurate PBR engine, Blender Cycles. Unreal Engine, a source-available product from Epic Games, is designed as a game engine that balances accuracy and performance to enable real-time rendering. Its latest release, Unreal Engine 5, introduces groundbreaking technologies that enable the rendering of highly complex scenes and lighting in real time, albeit with some approximations of the underlying physics. While they differ in computational and accuracy characteristics, both Blender Cycles and Unreal Engine 5 are powerful tools with high potential for simulating planetary surfaces. Along with their rendering engines, they offer procedural-generation capabilities that can be used to enhance existing planetary models or create entirely synthetic worlds. Their powerful technologies, coupled with the availability of their source code, are worth exploring to see how these tools could be employed for GNC simulations, particularly in the field of computer vision.

In the context of rendering planetary surfaces, two powerful and well-known commercial options are the Planet and Asteroid Natural scene Generation Utility (PANGU) developed for ESA by the University of Dundee[8] and SurRender developed by Airbus Defense and Space.[9] Previous work has presented additional end-to-end rendering pipelines for planetary bodies, such as the Interplanetary Rendering for Imaging and Sensors (IRIS) by Aiazzi et al.[10] and the Space Imaging Simulator for Proximity Operations (SISPO) by Pajusalu et al.[11] Unlike PANGU and SurRender, which are commercially available, IRIS is integrated within the Jet Propulsion Laboratory (JPL)'s DARTS/Dshell robotics closed-loop simulator. SISPO is an open-source tool leveraging Blender Cycles. However, unlike PANGU and SurRender, SISPO does not incorporate regolith-specific reflectance models like the Hapke or Lommel-Seeliger models, and relies on standard shaders that do not accurately capture the distinct properties of planetary surfaces. In this paper, we illustrate the use of Blender Cycles and Unreal Engine 5 to implement these regolith-specific reflectance models. This approach enables capabilities similar to those of the aforementioned commercial tools, but using source-available software. We also detail how to create fully procedural asteroids either using the highly accurate, but computationally demanding, path tracing method in Blender Cycles or the real-time rendering capabilities of Unreal Engine 5.

This work is divided into three main sections. In the first section, we provide an overview of the rendering process, with a particular focus on PBR, and elaborate on its primary components. We frame the rendering problem within the context of planetary surfaces, address the specifics of regolith-reflectance models with a qualitative description of the associated phenomena, and explain the fundamental PBR technique of path tracing. In the second section, we present our rendering pipeline and workflow in Blender Cycles, showcasing

*www.blender.org
†www.unrealengine.com

its effectiveness in rendering realistic celestial-body images from their 3D models as well as in generating fully procedural ones. In the third and final section, we explore the potential of Unreal Engine 5 for the real-time rendering and procedural generation of extremely high-resolution scenes. We conclude with a discussion on lessons learned and the limitations of these tools.

OVERVIEW OF PHYSICALLY BASED RENDERING

In 3D computer graphics, *rendering* typically refers to the process of generating a 2D image from a given 3D model of an observed environment, complete with light sources and an observer, or camera, that captures the scene or image. The output of the rendering process is influenced by: (1) the physical properties of the scene, such as its texture and reflectance, (2) the camera parameters, and (3) the type, direction, and intensity of the light source(s). Since the 1970s, a plethora of image-rendering algorithms have emerged, each with its own set of strengths and limitations.[12] As the field advanced and computational resources expanded, rendering techniques rooted in physical principles were proposed, where light is modeled as a flux that traverses and interacts with the observed scene. This approach, known as Physically Based Rendering (PBR), is capable of producing images nearly indistinguishable from real-life imagery—with the exception of certain lighting effects, such as refraction, polarization, and interference—provided that sufficient computational resources are available.[13] The following paragraphs delve into the main components of PBR.

The Rendering Equation

The effect of lighting and scene properties on the rendered output can be formally described by the *Rendering Equation*.[14,15] Consider a 3D surface illuminated by a light source. For an observing camera at a specific location, the appearance of such a scene depends on the interaction between the light and the surface, as well as the relative geometry among the camera, light, and surface. The Rendering Equation stipulates that the radiance departing from any point on the surface (L_o) is equal to the radiance emitted from that point (L_e) plus the total radiance reflected (L_r) from that same point. This is illustrated in Equation 1:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t)(\omega_i \cdot \mathbf{n}) d\omega_i \tag{1}$$

where $\mathbf{x}$ represents the 3D position of the point on the surface, λ denotes a specific light wavelength, and t is the time associated with the observation. $\mathbf{n}$ is the local surface normal, ω_i are the unit vectors from the point $\mathbf{x}$ toward the *i*ncoming light sources, which vary across the angular domain of integration; notice that ω_i captures both direct and indirect (e.g., reflections) light sources. ω_o are the unit vectors from the point $\mathbf{x}$ toward the *o*utgoing light directions. L_i is the irradiance (or incoming radiance) along ω_i. Ω is the unit hemisphere* representing the directional domain of all ω_o values, centered at $\mathbf{x}$ and such that $\omega_i \cdot \mathbf{n} > 0$. This condition implies that light is only reflected toward the space outside the local plane. Finally, f_r is the so-called *Bidirectional Reflectance Distribution Function* (BRDF), which is discussed in further detail in the next section. It is worth noting that the emission term L_e in Equation 1 can often be neglected for planetary-surface applications since they are not themselves emitting light, thus reducing the expression to the integral term L_r. In general, the point $\mathbf{x}$ can be influenced by both direct and indirect lighting (for instance, light reflected off another surface), and both of these components contribute to L_r.

The Rendering Equation adheres to the law of conservation of energy, as the total radiance for any surface point is conserved. The departing radiance L_o can be viewed as the equilibrium point in the expression and directly influences the rendering output by determining the local appearance of the scene. Consequently, in the rendering process, an estimation of L_o is needed for all parts of the scene that affect the captured image, including both direct and indirect lighting.

Although this model is widely employed in PBR applications, it remains a simplification of reality and fails to capture more complex light effects, such as transmission through media, polarization, or interference. Nevertheless, the Rendering Equation can enable photorealism for a wide range of rendered scenarios, including planetary surfaces.

*The double integral over the unit hemisphere Ω is here represented with a single-integral notation.

Bidirectional Reflectance Distribution Functions

The BRDF, f_r, describes the distribution of light reflected by a surface.[13] Given a point on the surface, the direction from that point to the light source (ω_i), and a departing direction of interest (ω_o), the function represents the ratio between the radiance reflected along ω_o and the irradiance received by the surface along ω_i, as shown in Equation 2:

$$f_r(\omega_i, \omega_o) = \frac{1}{L_i(\omega_i)\cos(\theta_i)} \frac{dL_r(\omega_o)}{d\omega_i} \tag{2}$$

where θ_i is the angle between ω_i and $\mathbf{n}$. During the rendering process, the BRDF is evaluated across the observed surface to compute the radiance departing from each surface element. Depending on the sampled outgoing direction ω_o, this radiance may be directed towards the observing camera, another surface element, or away from the scene entirely.

Similarly to the Rendering Equation, the BRDF also adheres to the conservation of energy, as

$$\int_\Omega f_r(\omega_i, \omega_o)\cos(\theta_o) d\omega_o \leqslant 1, \forall \omega_i \tag{3}$$

where θ_o is the angle between ω_o and $\mathbf{n}$. This class of functions is defined as "bidirectional" because it follows Helmholtz reciprocity, which states:

$$f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i) \tag{4}$$

BRDFs depend on the surface material and can vary dramatically. At one end of the spectrum are materials that produce perfect diffusion, appearing perfectly matte. In this case, the incoming irradiance is scattered equally in all directions of the unit hemisphere, i.e., scattering is isotropic. As a result, the output radiance depends only on the angle θ_i between the incoming light and the local surface normal, and not on the observer's viewing geometry. This scenario can be modeled using the Lambertian reflectance model, in which the BRDF is a constant value expressed as:[12]

$$f_{r,\mathrm{Lamb}} = \frac{\rho}{\pi} \tag{5}$$

where ρ represents the surface albedo, indicating the ratio of reflected radiance to the incident radiance; π is the normalization factor, resulting from the integration over the unit hemisphere, that ensures the energy conservation principle is upheld. Since $f_{r,\mathrm{Lamb}}$ is constant, the reflected radiance obtained from the rendering equation depends solely on the cosine term $\omega_i \cdot \mathbf{n}$ for a given surface. This is why the Lambertian model is often referred to as the *cosine law*.

At the opposite end of the spectrum are perfect mirrors, which reflect light in a manner where the incoming and reflected rays are symmetrical with respect to the local surface normal, and no light is scattered. Glossy materials fall somewhere in between: in these instances, light is scattered in a preferential direction (i.e., scattering is anisotropic). A combination of diffuse, glossy, and specular properties can often be used to produce realistic simulations of materials in PBR. However, accurately rendering planetary surfaces requires further physical considerations and more advanced models, as discussed in the next section.

Bidirectional Reflectance for Planetary Surfaces

Planetary surfaces are typically covered by *regolith*, a layer primarily composed of fine particles such as dust and small grains. One can conceptualize regolith as a layer of irregularly-shaped particles expanding from the surface towards the interior of the body. These particles, randomly oriented relative to each other, provoke repeated scattering of light within the medium, beneath the surface.[16] This unique material structure renders the common BRDFs, some of which have been introduced earlier, unsuitable for accurately modeling the reflectance properties of regolith. Several BRDFs have been proposed to simulate surface roughness, such

as the Oren-Nayar model, which assumes microscopic, randomly oriented facets, where each facet reflects light diffusely.[17] However, the reflectance of regolith is not fully captured by surface roughness alone, as it also involves subsurface scattering effects. Over the past several decades, multiple models have been developed and used to account for the fundamental physical principles of regolith reflectance. In this section, we introduce two state-of-the-art approaches: the Lommel-Seeliger model and the Hapke model. Although the derivation and comprehensive discussion of these models and the associated physical phenomena are beyond the scope of this paper, the summary below outlines their key principles and equations, as well as their relevance in PBR applications.

Lommel-Seeliger Model The Lommel-Seeliger model[16,18] is widely adopted in planetary photometry. Despite its relatively straightforward analytical formulation, it offers a robust approximation of regolith reflectance across most observing geometries. It is frequently used in applications involving both unresolved objects (for example, lightcurve inversion) and extended objects spanning multiple pixels (for instance, proximity optical navigation). This model treats regolith as a semi-infinite layer that light can penetrate. It operates under the assumption that each volumetric element in the regolith medium absorbs a portion of the incoming light and scatters the remainder isotropically. This process leads to the exponential attenuation of incoming light as it traverses the layer. Additionally, as the scattering is isotropic, only half of the incoming light is reflected towards the surface (with the other half directed towards the body's interior). This model only considers the first scattering event for each volume element, and is defined by Equation 6:

$$f_{r,\mathrm{LS}}(\omega_i, \omega_o) = \frac{w_0}{4\pi} \frac{1}{\mu + \mu_0}, \tag{6a}$$

$$\mu = \cos(\theta_o), \tag{6b}$$

$$\mu_0 = \cos(\theta_i) \tag{6c}$$

Here, w_0 denotes the *single-scattering albedo*, which is the probability that a photon, upon interacting with a particle, is scattered rather than absorbed, with $0 \leqslant w_0 \leqslant 1$. Since the Lommel-Seeliger model accounts for both the incoming and outgoing light directions, and models the light as scattered in all directions of the unit sphere, its normalization factor is 4π, in contrast to the Lambertian model which employs the π normalization factor.

A key property of the Lommel-Seeliger BRDF is its dependence on the angle between the observer and the sunlight (commonly referred to as the sun phase angle), encapsulated by the term $\mu + \mu_0$, as illustrated in Figure 1. For an arbitrary subsurface point, the magnitude of radiance reflected towards the observer hinges on the amount of light attenuation before reaching that specific point, and consequently on the depth of the point within the semi-infinite layer. The longer the path traveled by light through the regolith medium, the greater the light attenuation, and hence, the lower the radiance reflected from that point. In Figure 1, $0^\circ$and 90°sun phase cases are contrasted. In each scenario, two subsurface points are identified, each at a depth d, relative to the surface, along the observer's line of sight. One point is situated at local noon ($\omega_i = 0^\circ$) and one close to the terminator line $\omega_i \approx 90^\circ$. In the 0°-phase scenario, the reflected radiance for both points is equal, despite one point being much closer to the night side than the other, since the subsurface path traveled by the sunlight is also the same, resulting in an equal amount of light attenuation. On the contrary, in the 90°-phase scenario, the path traversed by the light to reach the point near the terminator is lengthier, and hence, the radiance reflected by that point is lower. This effect accounts for why planetary bodies covered by regolith appear as "flat" when viewed at a close-to-zero sun phase (for instance, a full Moon), while they appear more shaded at higher sun phases. This shifting appearance is a direct result of the variable light paths and corresponding attenuation, which are elegantly captured by the Lommel-Seeliger model.

One limitation of the Lommel-Seeliger model is that it treats regolith as a continuous medium, overlooking the fact that it is actually composed of discrete particles. While this approximation renders fairly accurate predictions for a wide variety of observing geometries, it becomes less precise as the sun phase angle approaches 0 degrees. This discrepancy arises because, in such conditions, the distinct effects of individual particles become more pronounced. To mitigate this issue, more advanced models, such as Hapke's, have been developed that take into account the discretized nature of the regolith medium.

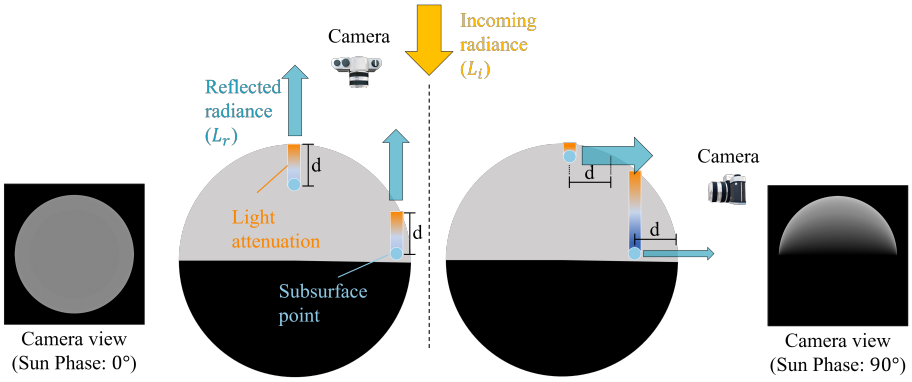


Figure 1: Illustration of the relationship between incoming radiance L_i and reflected radiance L_r as dictated by the Lommel-Seeliger model. Two sun-phase scenarios are depicted, considering subsurface points at a depth d. On the left, the sun phase angle is 0°. In this case, L_r is equal for all subsurface points at depth d. On the right, the sun phase angle is approximately 90°. Here, L_r depends on the angle ω_i between the incoming sunlight (related to L_i) and the local surface normal $\mathbf{n}$. The color gradient represents light attenuation, from warm to cold colors. The arrow width represents the magnitude of reflected radiance. Synthetic images of a sphere, generated based on the Lommel-Seeliger model, are also shown for each case. Figure adapted with permission from Kuzminykh.[19]

Hapke Model Hapke's model,[16] developed and refined over several decades, is often considered the gold standard for modeling reflectance from regolith surfaces. Unlike the Lommel-Seeliger model, which treats regolith as a semi-infinite continuous medium, the Hapke model acknowledges the discrete nature of regolith, composed of individual particles. It incorporates properties related to the particles and their structure, such as surface roughness, density, and porosity, and it accounts for multiple scattering events within the regolith layer.

The version of the Hapke model published in 2012 includes nine free parameters (detailed in Appendix A) and expresses its BRDF as shown in Equation 7:

$$f_{r,\text{Hapke}}(i,e,...) = \frac{LS(i_e,e_e)}{\mu_0} K \frac{w}{4\pi} [p(g)(1+B_{S0}B_S(g)) + M(i_e,e_e)]\,[1+B_{C0}B_C(g)]S(i,e,\psi) \quad (7)$$

In this equation, $LS(i_e,e_e)$ is the Lommel-Seeliger function, parameterized by the effective angles i_e and e_e; K is the porosity factor; w is the single-scattering albedo, the ratio of reflected to absorbed light for a particle; $p(g)$ is the Henyey-Greenstein double-lobed single particle phase function, describing the angular distribution of light scattered from a single particle, where g is the phase angle; B_{S0} and $B_S(g)$ represent the amplitude and function of the Shadow Hiding Opposition Effect (SHOE) respectively, which are discussed further in the next paragraph; $M(i_e,e_e)$ is the Isotropic Multiple Scattering Approximation (IMSA) function, capturing the effects of multiple scattering events; B_{C0} and $B_C(g)$ correspond to the amplitude and function of the Coherent Backscatter Opposition Effect (CBOE) respectively; and finally, $S(i,e,\psi)$ is the surface roughness (or shadowing) function, accounting for how microscopic shadowing effects between particles influence the macroscopic surface appearance, where i, e, and ψ are the incidence, emission, and azimuthal angles, respectively*. A detailed breakdown of the Hapke BRDF and its individual components is provided in Appendix A.

Unlike the Lommel-Seeliger model, which solely captures the macroscopic effects of the regolith, the Hapke model also accounts for the microscopic effects of the regolith particles. The most significant impact

*Notice that i and e are identical to θ_i and θ_o, defined previously. The notation difference is retained to preserve the common notation of the Hapke model.

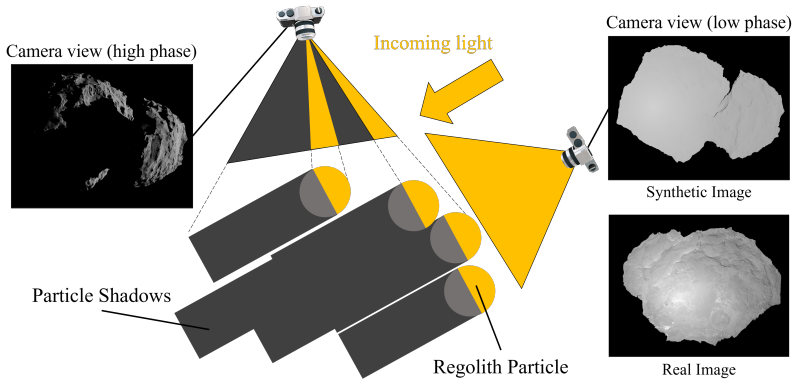


Figure 2: Illustration of the Shadow Hiding Opposition Effect (SHOE). When regolith is observed at a high sun phase, the microscopic shadows cast by the particles make the surface appear darker. Conversely, when observed at a zero or very low sun phase, these shadows are hidden behind the lit surface, resulting in a surge in brightness. The camera's field of view in the diagram represents the shadowed (grey) and lit (yellow) portions of the observed scene. Rendered images corresponding to these two scenarios are also shown, as well as a real image from the Rosetta spacecraft showcasing a real low-phase scenario (image credit: ESA). The synthetic images use a constant albedo throughout the surface. Figure adapted with permission from Kuzminykh.[19]

of the microscopic portion of the model is the opposition effect, or opposition surge,[16] which is caused by the discrete nature of regolith particles. This effect is observed as a sharp increase in reflectance at the surface point beneath the incoming light at low phase angles, i.e., when the Sun is positioned behind the observer. The Hapke model accounts for this through two separate components: CBOE and SHOE. The Coherent Backscatter Opposition Effect (CBOE) is a complex optical phenomenon that occurs in certain scattering mediums like planetary regoliths. The effect becomes significant when the particles on the regolith surface are of a similar size to the wavelength of the incoming light and are spaced more than one wavelength apart. Under these conditions, the incoming and scattered light waves can interfere constructively, enhancing the overall reflectance of the surface.

SHOE, on the other hand, can be described in more geometric terms. The individual particles comprising the regolith cast shadows when illuminated. While these microscopic shadows are typically unresolved to the human eye or a camera, they influence the macroscopic appearance of the surface, as depicted in Figure 2. At high Sun phases, these particulate shadows are visible from the observer's viewpoint and darken the surface due to the shadows they cast on neighboring particles. Conversely, at zero Sun phase, the shadows are concealed behind their respective particles, resulting in an observer's viewpoint dominated by the lit portions of the particulate. This causes the surface to appear much brighter than it would in a continuous medium, as is assumed in the Lommel-Seeliger model.

Figure 3 displays a comparison of synthetic images produced using the Hapke, Lommel-Seeliger, and Lambertian models at various sun phase angles. As expected, the Hapke model uniquely captures the opposition effect, resulting in a marked brightness surge at a zero sun phase. Conversely, the overall reflectance predicted by the Lommel-Seeliger and Lambertian models is considerably lower under the same conditions. It is also worth noting that the Lambertian model fails to capture the flattening of the surface appearance observable at low sun phases, as it simply describes reflectance via a cosine law. However, as the sun phase angle nears 90 degrees, the reflectance outputs from all three models start to converge.

Path Tracing

Solving the Rendering Equation (Equation 1) is the overarching challenge in image rendering. Given an observed scene illuminated by various light sources, the goal is to compute the interaction of light with the

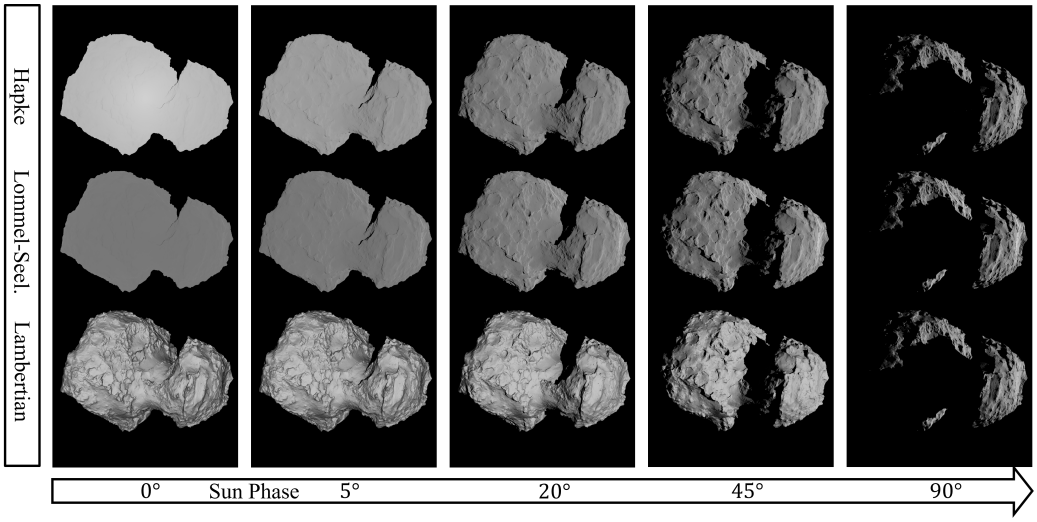


Figure 3: Comparison of the Hapke (top row), Lommel-Seeliger (middle row), and Lambertian (bottom row) BRDFs at different sun phase angles and fixed camera exposure, rendered using Blender Cycles and the Open Shading Language. The celestial body represented in these renderings is Comet 67P Churyumov-Gerasimenko. The parameters used for the reflectance models are reported in Table 1. The image exposure for the Hapke and Lommel-Seeliger models is set to the same value. For the Lambertian model, however, we manually adjust the exposure to ensure that it is comparable to the other two models in the 90-degree scenario.

scene in order to recreate the scenario as it would appear to the observer. In reality, this process involves an enormous number of photons, which are nearly impossible to simulate numerically in their entirety. As a result, all rendering approaches require some level of approximation.

Various techniques exist within the realm of PBR that aim to emulate light behavior to generate photorealistic renders while managing the computational complexity. These include ray casting, ray tracing, path tracing, and radiosity. Ray casting is the simplest technique, where rays are sent from the camera, through each pixel, into the scene, and only the closest object the ray intersects determines the pixel value. Ray casting is computationally efficient, but does not simulate shadows or indirect lighting effects resulting from multiple light reflections. In ray tracing, rays are also sent from the camera into the scene, but when they hit surfaces, additional rays are sent out to simulate how light would be reflected by any specific surface; while this allows capturing shadows and reflections, it typically only considers direct light interactions (i.e., light only bounces once before reaching the camera). Path tracing is an extension of ray tracing, where multiple bounces on the surface are simulated, thus increasing photorealism and simulating indirect lighting. Radiosity is different from the aforementioned techniques in that is based on surface energy exchange. It computes the surface brightness by accounting for the indirect lighting from the entire scene and particularly excels at simulating soft, diffuse lighting, in contrast with shiny surface. Radiosity is computationally expensive, compared to other techniques. each technique comes with its own advantages, drawbacks, and ideal use-case scenarios. The ultimate goal is to simulate *global illumination*, that is, the complete range of lighting conditions comprised of both direct and indirect lighting—the latter being light that is reflected off other surfaces within the scene.

Among these methods, path tracing is often considered the gold standard in terms of photorealism, despite its higher computational cost. Other methods often build upon the principles of path tracing, introducing approximations to reduce computation. Given that path tracing is at the heart of the Blender Cycles rendering engine and has a foundational role within rendering techniques, it deserves a more detailed discussion, which is provided in the subsequent paragraphs.

Path tracing, as the name suggests, is based upon the propagation of light paths throughout the scene, which

in our case is composed of some 3D surface. This method propagates a set of random light rays backwards from the camera through the focal plane to the surface and reflected off the surface until they reach the light source. The rationale for propagating light rays backwards—starting from the camera rather than from the light source—is to avoid unnecessary computation. All rays starting from the camera necessarily affect the image's appearance, whereas those starting from the light source do not necessarily reach the camera. Rays that do not reach the light source, either because of occlusions or reflection away from the light source, are discarded. The remaining valid set is then repropagated forward from the light source to surface and then to the camera to determine how light radiance is affected by interactions with the scene. The image is then rendered by computing the path of each individual light ray and averaging its effects for each pixel in the image.

Figure 4 illustrates a simplified path tracing process, where a single light source is present, and surface reflections are the only effect considered (as opposed to also surface emissions and volumetric effects). First, a set of light rays is generated originating from the camera's focal point and crossing the image plane. Each pixel in the image can be sampled by one or more rays; typically, multiple rays (e.g., tens to hundreds) per pixel are required for accurate results, especially in complex scenes or to avoid aliasing. Next, each ray is tested for intersection with the surfaces in the scene. If an intersection is found, both direct and indirect illumination effects are typically computed at the intersection point by evaluating the surface BRDF. Direct illumination is computed by tracing a ray (known as a shadow ray) from the surface point directly to the light source. If this path is blocked by the scene, then the point is in shadow relative to the light source and does not have direct illumination. Conversely, if there is no obstruction, the direct illumination on that surface point is computed and accounted for in the rendered image. Indirect illumination is evaluated by randomly sampling a bounce direction ω_i from the BRDF, which determines a single instance of light-ray reflection at the surface intersection. It is important to note that because path tracing propagates light in reverse, the input here is ω_r and the output is ω_i. The process of finding and evaluating intersection events is repeated until the bouncing ray either encounters the light source, reaches a user-set maximum number of bounces, or leaves the scene. Rays that leave the scene do not contribute to the final rendered image and are discarded.

Once all the paths from the camera to the light source are computed, each path is re-evaluated in reverse—from the light source to the camera to determine how light radiance is affected by interactions with the scene. This step involves re-evaluating the BRDF at each surface intersection to account for light attenuation and indirect lighting effects, such as changes in light color or intensity, or both, as it bounces multiple times. It is important to note that light paths can be computed from the camera to the light source, or vice versa, due to the principle of reciprocity of BRDFs (as shown in Equation 4).

Consequently, each ray will have an associated color value when it reaches the camera. The output color for each pixel is determined by averaging the colors of all the rays within that pixel, resulting in the final output image.

Path tracing, thanks to its faithful emulation of light transport through a scene, provides unbiased rendering results and is capable of simulating global illumination. As a Monte Carlo method, the accuracy of path tracing increases with the number of samples per pixel and the number of light bounce instances, potentially enabling the production of images indistinguishable from reality. This makes it the method of choice for applications where photorealism is the priority. However, path tracing is a computationally intensive process, demanding significant resources to accurately render complex scenes. To improve efficiency and realism, several implementations and enhancements have been proposed, including Importance Sampling and Metropolis Light Transport, both favoring light paths that contribute more significantly to the final pixel color, and Bidirectional Path Tracing, where light rays are traced from both the camera and the light sources.[13] Lastly, it should be noted that most path-tracing implementations simulate light transport as instantaneous, thus overlooking the finite speed of light. This assumption can be insufficient when the camera is far from the observed scene or in applications that require high-precision rendering. In such scenarios, accounting for light travel time in the path-tracing implementation may be critical.

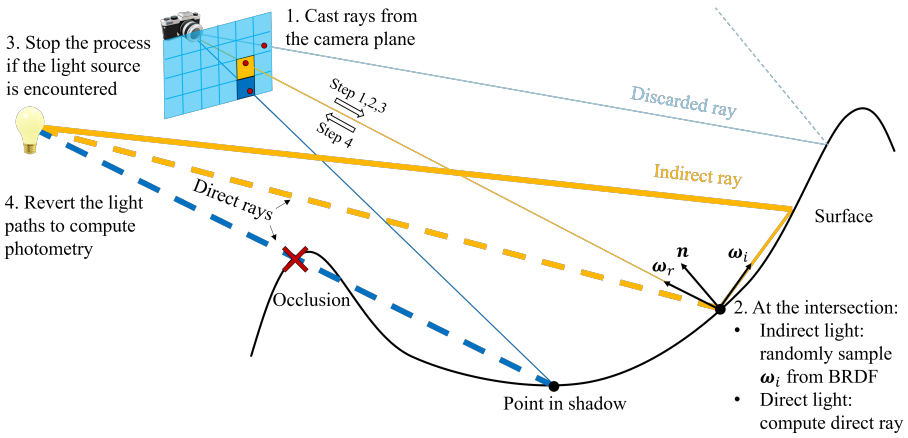


Figure 4: Path tracing process in a scene with a single light source, considering only reflections. Rays from the camera intersect with surfaces in the scene. At the intersection points, both direct and indirect light is computed. Indirect lighting is computed by propagating random light bounces through the scene. The indirect-light propagation terminates when a ray meets a light source, exits the scene, or exceeds a set number of bounces. Ray thickness indicates the light-radiance magnitude, typically decreasing with each interaction. Ray colors match the affected pixel colors; yellow represents lit rays, while dark blue signifies shadow rays, which are a result of light being occluded by the scene. Arrows depict the path-tracing direction for steps 1-3 (from camera to scene), contrasting with step 4 (from scene to camera).

PHOTOREALISTIC RENDERING USING BLENDER CYCLES

Blender is a widely-used, open-source software for 3D computer graphics. Originally developed as an artistic tool for domains such as animation, visual effects, and video gaming, it has since seen increased adoption in scientific and engineering applications, including space imagery, thanks to its physics-based rendering engine, Blender Cycles. Supported by a robust community, Blender boasts extensive learning resources available for both users and developers*. While this work does not cover the artistic aspects of the tool, it focuses on features and use cases related to PBR for planetary surfaces. These include Blender's physics-based rendering engine, camera modeling, procedural terrain generation, and programmable BRDFs. Here, we discuss using Blender through its Graphical User Interface (GUI) and Python Application Programming Interface (API). Finally, we present the rendering setup used in our research and showcase some example results.

Cycles Rendering Engine

Cycles is a high-performance rendering engine fully integrated within Blender†. As an unbiased rendering engine, Cycles adheres to an accurate simulation of light transport without simplifications that could bias the result. It is based on path tracing, which accurately simulates phenomena such as reflections from multiple surfaces. This allows Cycles to simulate global illumination even in complex scenes—for instance, those involving multiple light sources or repeated surface scattering. The ability to produce photorealistic images is further bolstered by features that allow the creation of complex, custom materials. These include the Node Editor interface and the Open Shading Language (OSL) for custom BRDFs, both of which are described in subsequent sections. While Cycles supports both CPU and GPU computing, GPU computing is not supported when using OSL. Despite its relatively high computational cost, Cycles is considered a leading rendering engine in both the artistic and scientific domains due to its accuracy and flexibility.

*https://developer.blender.org/
†https://docs.blender.org/manual/en/latest/render/cycles/

Blender's GUI and Python API

In Blender, two main interfaces are available: a GUI and a Python API. Although the same tasks can generally be accomplished through both, each interface is best suited for specific use cases. The GUI is a powerful tool that provides a visual aid for users, particularly for tasks such as manually editing a surface mesh or manipulating objects within a scene. On the other hand, the Python API is especially useful for the automation and execution of rendering pipelines. It allows for streamlined processes including loading data (e.g., mesh files and camera parameters), updating the camera pose, and rendering images in a loop. The subsequent section delves into more detail on how we used these tools within our rendering process.

Rendering Workflow

The workflow that we use to set up rendering simulations in Blender is divided into two steps. First, a rendering setup with certain "locked" settings is defined through the GUI and the Node Editor (described in the next section). These settings are those considered constant for a specific analysis, such as planetary-surface meshes, albedo and elevation maps, material BRDFs, as well as light properties, like sunlight intensity. These settings are saved in a "startup" file (with a *.blend* file extension), which opens a Blender project with the specified settings. This file can be run either using the GUI or via the terminal.

The second step involves the execution of a Python script within the Blender environment, using the Blender Python (*bpy*) API. This powerful interface allows for programmatic manipulation of nearly all aspects of Blender, including setting and retrieving properties of simulated objects and importing and exporting data. In our case, the Python script sets the simulation-specific parameters, such as the camera position, orientation, and intrinsic parameters, the planetary body or surface of interest, and the sunlight direction, as well as rendering-specific parameters such as imaging resolution and path-tracing settings. Notably, Blender can auto-generate the Python commands corresponding to tasks performed by the user via the GUI. This feature can simplify the process of writing Python scripts for Blender. While it is possible to automate most or all tasks using Python, we found graphical tools to be more practical for certain operations, such as customizing the material properties via the Node Editor, which we discuss next.

Node Editor: Setting Material Properties

A node-based system is a graphical representation of a data flow, where each node represents a particular operation, function, or data source; examples include a mathematical operator, a texture file, or a BRDF model. The Node Editor is a Blender GUI that allows users to create, customize, and edit complex node-based systems. These systems can represent various aspects of the rendering setup and 3D scene, such as material surfaces and volumetric properties, textures, lighting, and even procedural models. These nodes can be connected to one another to form a network, where the output of one node can be used as the input to another. This allows for a high degree of flexibility and complexity in defining various aspects of a 3D scene.

In the context of planetary-image rendering, the node-editing system is particularly useful to define the surface-appearance models. As an example, the node-system instance used in this work to render a Moon image is shown in Figure 5. In this case, there are four node groups required to simulate the surface appearance: the observing geometry, the surface BRDF, the surface albedo map (usually referred to as *texture* in computer graphics), and the surface displacement model. The first group consists of the observing-geometry nodes, which store the camera position and light direction, used for runtime computation of ω_r and ω_i, respectively. These direction values can be obtained using Blender's *drivers*, tools to automatically query parameter values from simulated objects. In this way, the values in the nodes for ω_o and ω_i are automatically updated at runtime. The second node group contains the BRDF-related node. As discussed in the following paragraph, the BRDF node can be programmatically defined using OSL. This node is linked to the OSL script defining the reflectance function, and is connected with the BRDF's inputs and outputs such as the sunlight and camera direction vectors previously introduced. The third node group is related to the albedo map, which is another input to the BRDF node. The albedo node usually contains surface-albedo data in the form of an image. When irregular 3D shapes are used, a process called *UV unwrapping* is typically needed in order to map the albedo information onto the surface of a 3D model; UV unwrapping is not used in this work. The fourth node

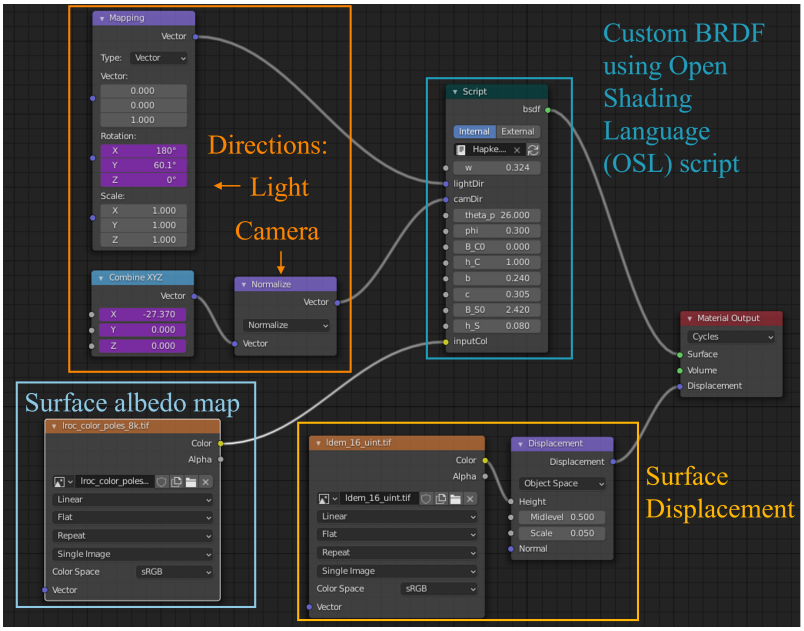


Figure 5: Screen capture of the Blender Cycles Node Editor GUI, used to generate the Moon render in this study (Figure 7). The nodes composing the data flow can be grouped into four categories: observing-geometry information (top-left), BRDF (top-right), surface-albedo information (bottom-left), and surface-displacement information (bottom-right). Blender's *drivers* are used to query the viewing and light directions at runtime.

group regards surface displacement, which provides additional 3D surface refinement of the original mesh if the latter is not sufficiently dense. The displacement data can either be provided by the user, e.g. in the form of a Digital Elevation Model (DEM), or be procedurally generated, as described in a following section.

This node system is connected to the material output, ensuring the defined relationships are used by the rendering engine during the rendering process. This means that the textures, shading, and other material properties defined by the nodes are dynamically evaluated and applied to the 3D object at runtime.

Programmable BRDFs using Open Shading Language

Open Shading Language (OSL) is a programming language specifically designed for crafting shaders, including those not natively incorporated in Blender Cycles.[20] Shaders in computer graphics are mathematical models used to determine the appearance of a 3D surface or volume. Thus, a BRDF can be thought of as a specific component of a shader dedicated to defining surface appearance. The syntax of OSL shares similarities with C and C++, but it includes shader-specific variables such as color, position, and surface normal. These variables are used in computations and to return the output in the appropriate format. This language allows users to define BRDFs, as well as their inputs, programmatically. Then, Cycles' rendering engine uses the OSL code to compute the surface appearance during the path-tracing process. For realistic rendering of planetary surfaces, which often requires regolith-specific reflectance models, OSL is a valuable tool. In this work, we use OSL to implement the Lambertian, Lommel-Seeliger, and Hapke BRDFs (as shown in Figure 3).

Procedural Terrain Generation

Procedural generation is a method in computer graphics where content is generated from an algorithm (or program) possibly injected with randomness rather than using pre-existing models and data. Blender Cycles provides a comprehensive toolkit for the procedural generation of material properties, such 3D displacement

and textures. These features can be used to either augment surface details of an existing 3D model or to create entirely synthetic 3D objects, e.g., artificial asteroids.

Procedural materials can be implemented through the node system in Blender's Node Editor. A typical node system for procedural generation usually includes two types of nodes: noise models and mathematical operators. Noise models serve as the foundational functions or algorithms that create the building blocks of the procedural output. They can be manipulated and combined to produce the desired output. Two common noise models are Perlin noise[21] and Voronoi (or Worley) noise,[22] which can simulate natural-looking topography and crater distributions on planetary surfaces, respectively. Mathematical operators, also represented by nodes in Blender Cycles, are then applied to such noise elements. These operations can involve summing, multiplying, or applying other transformations to the noise functions.

The output data from procedural generation is often stored as a 2D image, where the picture elements represent an albedo or elevation map—with the latter dictating the local surface displacement. There are primarily three methods to achieve surface displacement: true 3D displacement, bump mapping, and micropolygon displacement, all of which may be employed individually or synergistically.

A bump map is a technique that creates the illusion of depth on the surface of a 3D object without the need for additional polygons.[12] This illusion is achieved by manipulating the surface normals to emulate the desired textural detail. This approach is computationally efficient and useful for representing small-scale details on a surface without significantly increasing the complexity of the model.

In conventional 'true' displacement, the mesh elements shift in accordance with the input data. However, the resolution of the surface mesh becomes a limiting factor for the quality of the surface appearance. In contrast, micropolygon displacement is a technique that adaptively subdivides the surface elements of a 3D object, such as mesh triangles, into smaller components known as micropolygons.[23] These micropolygons are then individually displaced based on the displacement map. The degree of subdivision, and thus the level of detail, can be dynamically adjusted based on factors such as the distance to the camera and the resolution of the final render. This ensures that the model's complexity and computational requirements scale appropriately to the rendering context, maximizing detail while minimizing unnecessary computational load. This approach is especially beneficial for rendering high-detail close-up views, such as those required for proximity operations and landings.

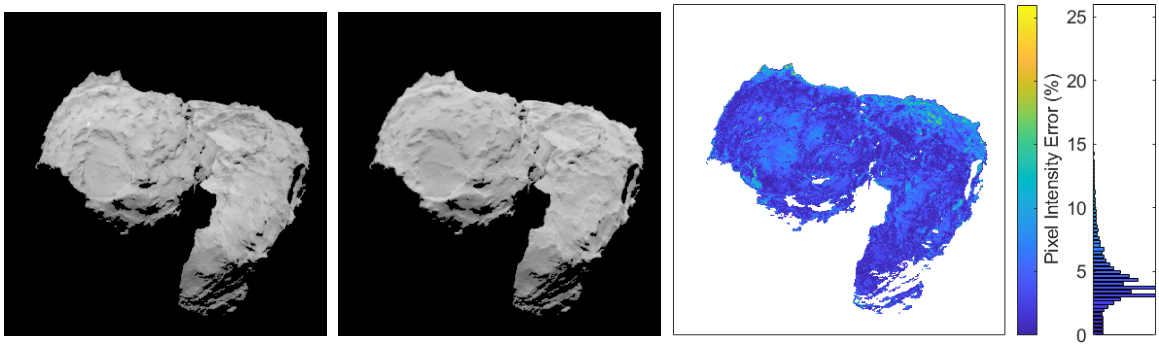
Constructing a procedural node system often necessitates initial manual crafting and parameter adjustment to attain the desired output. However, once this system is established, it enables the generation of an indefinite number of procedural outputs. This can be accomplished by programmatically altering the random seed or randomly sampling the model parameters, thus offering a robust and flexible way to generate diverse outcomes.

Camera Model

Blender Cycles provides a variety of camera settings and parameters that closely replicate those of real-world cameras. Key elements such as focal length, sensor size, depth of field, distortion types, and shutter speed are available, with the last being particularly useful to simulate image smearing.

A notable limitation in Blender's camera model pertains to its handling of exposure settings. While Blender does allow adjustments to image exposure mimicking that of real-world cameras, its exposure parameter lacks a direct physical analog. This discrepancy originates from Blender's image generation method—path tracing—which relies on path samples instead of actual photon fluxes. Consequently, the exposure parameter in Blender doesn't translate directly to a physical exposure time. Additionally, Blender doesn't incorporate a camera Point Spread Function (PSF) to emulate the camera's response to a point light source. Integrating such models and parameters into Blender's rendering pipeline would demand additional work.

While Blender cannot simulate error sources such as read noise, quantization, cosmic rays, and dark current, these effects are typically added in a post-processing stage without loss of realism.



(a) Real image (foreground) (b) Synthetic Image (foreground) (c) Foreground intensity errors

Figure 6: Comparison between a real and synthetic image of Comet 67P, captured from Rosetta's NAVCAM. The background is eliminated from both images using the same mask. The synthetic image is produced using Blender's Cycles rendering engine, along with the Hapke BRDF implemented in OSL. Also reported are the pixel intensity errors between the images, along with their respective error histogram. Errors are exclusively displayed for the image foreground, as the background is neither simulated nor utilized for histogram equalization. Images are cropped for simplified visualization.

Rendering Results

In this section, we demonstrate the capabilities of Blender Cycles for procedural generation and rendering. We showcase the use of this tool to both reproduce real imagery, using the image metadata, and generate fully procedural objects, which may be used for algorithm training and testing.

Reproduction of Real Imagery. To evaluate these modeling tools, we reproduce an image of a complex planetary body and compare it to real one acquired from a spacecraft. Here we reproduce an image of comet 67P Churyumov-Gerasimenko taken during the Rosetta mission*.[24] The image, captured by the Navigation Camera (NAVCAM) on August 5, 2014, was taken approximately 153 km from the comet nucleus. To replicate the observation geometry and set the camera properties, we use the SPICE kernels associated with the image metadata, provided by NASA's Navigation and Ancillary Information Facility (NAIF)†. We use the meter-level shape model of comet 67P from Preusker et al. to simulate the surface shape.[25] We render the image based on several assumptions: we ignore geometric distortion and any other image error sources, neglect image smearing (noting that the actual integration time is 3.0 seconds), and assume a constant albedo across the comet surface. We use our OSL implementation of the Hapke model, and the Hapke parameters retrieved by Davidsson et al.,[26] which are listed in Table 1. Since, as previously mentioned, Cycles does not employ a physical exposure time for rendering images, we manually adjust Blender's exposure parameter to avoid saturation. Then, we use histogram equalization[27] to match the intensities of the two images. Given that the background of the synthetic image possesses a perfectly zero intensity—owing to the aforementioned assumptions—we identify and eliminate the real image's background utilizing Otsu's threshold selection method.[28] Subsequently, we apply histogram matching solely to the image foreground using a binary mask‡.

Figure 6 displays both the real and synthetic images, with the background removed using the same mask for each. It also includes the per-pixel intensity errors between the two images, along with the associated error histogram. The relative error primarily spans lower values, with the 95th percentile of the error distribution at 8.7% and an average value of 3.5%. It is important to note that these specific error magnitudes are also contingent upon the Hapke parameters utilized during rendering.

*https://archives.esac.esa.int/psa/

†https://naif.jpl.nasa.gov/pub/naif/ROSETTA/kernels/

‡After simulating the imaging geometry using the SPICE kernels, we observe an image translation offset of approximately 5 and 3 pixels in the horizontal and vertical image directions, respectively. Consequently, to fully align the synthetic image and the real one, we had to conduct a manual image-registration step.

w	θ_p	ϕ	B_{C0}	h_C	b	c	B_{S0}	h_S
0.055	16	0	0	0	-0.456	1	1	0.035

Table 1: Hapke parameters from Davidsson et al.,[26] used in Blender Cycles to reproduce the Rosetta NAVCAM image in Figure 6b.

Figure 7: Render of the Moon using Blender Cycles, based on real elevation and albedo data from NASA's Scientific Visualization Studio. The render makes use of our Hapke model OSL implementation with parameters derived by Sato et al.[29] The observation geometry and phase angle are arbitrary.

Augmentation of Shapes with Albedo and Elevation Maps To test the application of albedo and height maps, we use the Moon as a case study. Starting with a spherical shape discretized into triangular elements, we further subdivide it to improve smoothness and employ micropolygon displacement. Subsequently, we construct a node system (illustrated in Figure 5) where the albedo map is fed into the BRDF, to evaluate various albedo values across the surface. Concurrently, the elevation map is linked to the surface displacement node, enabling the rendering engine to evaluate it in real time. We use the lunar albedo and elevation maps provided by NASA's Scientific Visualization Studio*. This render also relies on our Hapke model OSL implementation and leverages the Moon's Hapke parameters as derived by Sato et al.[29] The resulting Moon render can be observed in Figure 7. The observation geometry and phase angle are arbitrary in this instance.

Generation of Fully-Procedural Small Bodies To create entirely artificial celestial bodies, we use the procedural generation capabilities of Blender Cycles with the previously described node systems. In this instance, we fabricate a large asteroid, characterized by a rich field of craters, using procedural nodes to randomly generate topography and crater populations. As in the case of the Moon rendering, we initiate the body as a discretized sphere (we could have alternatively used other shape primitives, such as ellipsoids or existing planetary shape models), which we further subdivide and subject to micropolygon displacement. Subsequently, we construct a node system (completely illustrated in Figure 10, in Appendix B) as follows.

We construct a node subsystem that generates Perlin noise functions at three distinct scales. The noise data are then merged and stored as a single 2D image. These three layers are then manipulated and integrated using a variety of nodes executing mathematical operations. This multi-scale Perlin noise is used to simulate multi-scale surface topography. In a similar fashion, we apply a node subsystem that combines multiple layers of Voronoi noise (also stored as image data) to simulate the crater population. The Perlin and Voronoi noises are then combined to create a unified noise image. This data can serve as both an albedo and an elevation map. The noise image is then input to the BRDF to simulate a randomized albedo distribution, and to the displacement node to produce surface reliefs by displacing the original spherical surface. We use the Hapke BRDF implemented in OSL to evaluate reflectance. It is worth mentioning that the noise parameters we select throughout the node system serve purely demonstrative purposes and are not based on physical models of planetary topography and crater populations.

Figure 8 displays two instances of fully-procedural asteroids, each produced using slightly different param-

*https://svs.gsfc.nasa.gov/cgi-bin/details.cgi?aid=4720

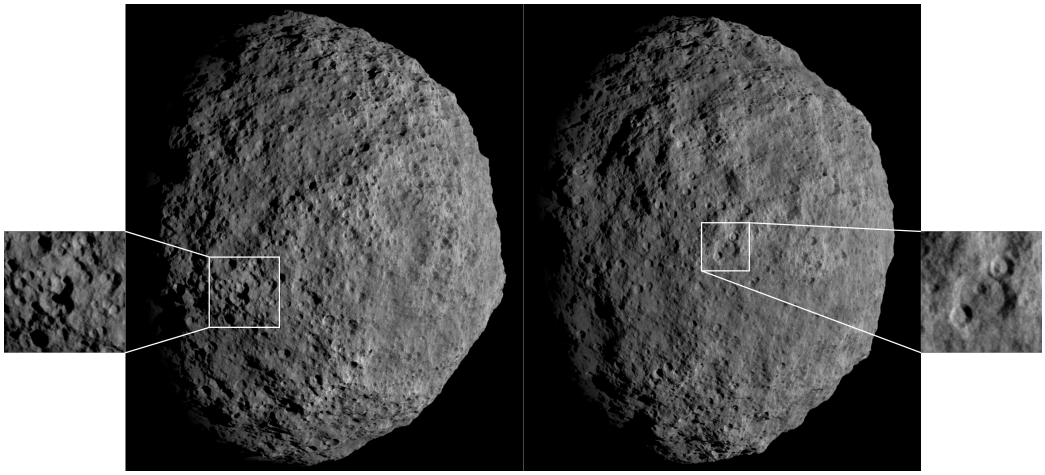


Figure 8: Two examples of entirely procedural asteroids, created starting from a sphere and defining the surface displacement and albedo based on a combination of Perlin and Voronoi noise functions. Close-up views of the surface are also provided. The scenes are rendered in Blender Cycles, using a Hapke BRDF implementation in OSL.

eters. Properties such as crater size, density, depth, as well as topography roughness and amplitude can be manipulated by adjusting the corresponding parameters in the node system. As these renders use path tracing and true surface displacement, the shadows cast on the surface are physically accurate, contingent upon the 3D geometry.

While node systems can be arbitrarily complex and tailored to the intended purpose, this initial case study implies that procedural techniques bear potential for generating realistic renders of planetary surfaces. A significant advantage of procedural methods lies in their ability to quickly generate an infinite variety of unique and novel shapes, featuring characteristics representative of actual celestial bodies, once the node system is established. Furthermore, it is important to note that rock-like features could also be generated procedurally or scattered using Blender's particle systems, as shown by Pugliatti and Topputo, for example.[30]

UNREAL ENGINE 5

In this section, we introduce and examine Unreal Engine 5 (UE5). Unlike Blender Cycles, a PBR engine designed for precise, offline computation, UE5 is a game engine optimized for real-time rendering. It facilitates the creation of visually impressive, photorealistic scenes but introduces some approximations to the underlying physics involved in the rendering process, compared to path tracing. Ground-breaking technologies introduced by UE5 enable real-time rendering of exceedingly complex scenes with millions of surface elements, enabled by an optimized surface representation which adaptively accounts for the observer's viewpoint, *Nanite*, and an efficient lighting technique to compute global illumination in real time, *Lumen*[*]. These advancements represent significant progress in the field of CGI and are likely to impact a variety of applications profoundly.

Unlike Blender, which is open-source, UE5 is source-available software. This means that while users can access the source code, the software is subject to different licensing restrictions and is not completely open for modification and redistribution under the same terms as open-source software.

Interestingly, Unreal Engine 5 (UE5) shares several features with Blender Cycles, including the implementation of realistic camera models, procedural generation capabilities, programmable BRDFs, and a Python API (though UE5 also offers a C++ API). Despite specific nodes and workflows varying between the two platforms, the underlying principles remain strikingly similar. For instance, UE5's Material Editor can be

[*]`https://docs.unrealengine.com/5.0/`

employed to build intricate node systems for procedural terrain generation, much like in Blender Cycles. In UE5's case, the High-Level Shading Language (HLSL) replaces OSL for programming custom BRDFs. While OSL is mostly used in offline rendering, such as in the CGI and film industries, HLSL supports GPU computing and is commonly used for real-time rendering, such as in video gaming.

Given the similarities between Blender Cycles and UE5 functionalities—despite their aforemenentioned differences—we will not delve deeper into discussing UE5 implementations.

Nanite

Nanite, a technology introduced with UE5, has brought an unprecedented level of detail to the field of CGI. It enables the efficient representation and rendering of 3D geometry, permitting the creation of scenes with millions of polygons—a feat previously unachievable in real-time rendering. Nanite is engineered to automatically update the scene's level of detail in real time. It allows users to import 3D objects of arbitrarily high resolution into the scene, then automatically adjusts the level of detail based on what the camera can visualize at a given moment. This is accomplished through several technological advancements, including virtualized geometry to more efficiently represent complex forms, hierarchical clustering of the original geometry, and efficient data streaming into the GPU.

In the realm of planetary surface generation and rendering, Nanite drastically reduces the computational cost of large shape models, allowing simulation of entire planetary bodies, possibly with centimeter or millimeter-level resolution. Consequently, it enables seamless simulations of mission scenarios that involve substantial changes in scale, such as flybys, approach, and landings on planetary bodies.

Lumen

Another significant innovation brought about by UE5 is Lumen, a real-time global illumination system capable of rendering both direct and indirect lighting within a scene. This system produces high-fidelity renders even in scenarios involving complex scenes and lighting.

To achieve this, Lumen first creates a surface cache for every object within the scene, storing essential material properties such as albedo, reflectance, and surface normals. The system then uses a combination of screen space tracing*, path tracing, and importance sampling to strike an optimal balance between accuracy and performance. Lighting computations are accurately performed for objects within the camera's field of view, while a Signed Distance Field (SDF) voxelization technique approximates lighting emanating from outside the view. Rays traced through this voxel grid are used to compute global illumination. Additionally, Lumen leverages spatial filtering (computation conducted on nearby pixels) and temporal filtering (computation conducted in previous frames) to reduce computational costs and enhance accuracy.

Accurate lighting and global illumination are essential components of rendering planetary surfaces, especially those exhibiting rugged topography. These often require offline techniques, like path tracing, where the computational demand scales significantly with the resolution and complexity of the scene. On the other hand, Lumen enables real-time rendering of very complex lighting conditions, using the aforementioned techniques. This makes Lumen a pivotal tool for planetary applications.

Generating Procedural Rubble-Pile Asteroids

In this section, we explore the potential of UE5 in producing realistic renders of planetary surfaces. To illustrate both its terrain generation and real-time rendering capabilities, we procedurally generate a rubble-pile asteroid composed of thousands of individual rocks—amounting to billions of surface elements—and render it in real time.

The asteroid generation process is divided into two steps: (i) the import of a global, low-resolution 3D model to define the asteroid's macroscopic shape, and (ii) the procedural generation of a high-resolution

*Screen space tracing is a computationally efficient technique to simulate indirect lighting by tracing rays in the 2D screen space, to approximate the 3D scene. This is done using the information already available in the so-called frame buffer, containing data such as pixel color and depth, without simulating the actual bounces through the 3D scene.

layer of rock elements, scattered atop the low-resolution surface. We start with a shape model from asteroid Ryugu—a rubble-pile asteroid explored by JAXA's Hayabusa2 mission—to establish the low-resolution shape.[2]

Next, we select a group of rocks from Megascans*, a comprehensive library of high-quality 3D scans. These are real-world materials and objects, primarily obtained via photogrammetric shape reconstruction. From the catalog, we select ten different rock instances, each comprising approximately one million triangular elements. These selected instances represent the base assets used to generate the random population and are thus set as "foliage meshes." In UE5, foliage refers to the system enabling the scattering of a large number of assets throughout the scene. Following this, we create procedural-foliage spawners (PFS), tools designed to generate foliage in the scene automatically and algorithmically based on predefined rules and parameters. We configure PFS instances to generate the rock population starting from a base of ten unique rocks. These rocks are then automatically randomized in terms of placement (position and orientation) and size within the set parameters and following predefined curves to emulate a more realistic rock distribution. Finally, we set the PFS bounding volumes around the asteroid surface to define the area subject to the procedural rock layer. Once the PFS parameters are defined, the procedural process can be automated and the scene is updated when the PFS parameters are changed. Note that the rocks selected for this demonstration are terrestrial, as indicated by their features, and are used solely for demonstration purposes. Furthermore, the size frequency distribution is manually set and does not adhere to any real distribution of rocks on asteroids (though this could be achieved using PFS parameters). Lastly, we do not set any custom BRDF for this render, and instead use the pre-configured Megascans material associated with the photogrammetric rock scans. These compute lighting based on the rocks' material data such as albedo, normals, and roughness maps.

Rendered results for the procedural rubble-pile asteroid are displayed in Figure 9, at varying camera distances from the asteroid surface. These images are real-time screen captures of the viewed scene, rendered on a laptop as the user navigates through the scene at multiple frames per second. It is noteworthy that some rendered scenes include at least billions of triangles. The camera exposure is adjusted automatically on the fly. Indirect lighting effects become prominent when the camera approaches the surface—where shadows are not as sharp—showcasing the remarkable capabilities of UE5 for global illumination.

CONCLUSIONS AND FUTURE WORK

This work explores the utility of source-available tools, specifically Blender Cycles and Unreal Engine 5, for rendering planetary surfaces and procedurally generating synthetic celestial bodies. While we quantitatively analyze Blender Cycles' accuracy, by comparing its rendering output with a real image from a spacecraft, the Unreal Engine 5 capabilities are demonstrated without quantitative metrics—which we will address in future work. Both platforms excel in producing visually compelling imagery, supporting programmable reflectance functions such as the Hapke model and offering robust tools for procedural terrain generation. Blender Cycles is designed for offline, high-fidelity rendering using path tracing, while Unreal Engine 5 prioritizes real-time rendering through its innovative Nanite and Lumen technologies. The choice between the two primarily depends on specific needs, balancing factors such as accuracy, computational cost, and application. Generally, Unreal Engine 5 presents a more versatile tool. Its real-time global illumination rendering of scenes encompassing billions of triangles marks a significant shift in the rendering field, albeit at the cost of certain physical approximations. In addition to its computational advantage, Unreal Engine 5 facilitates the creation of highly detailed procedural worlds, employing not only mathematical functions but also realistic 3D shapes like photogrammetric rock scans. Such capabilities herald new opportunities for the validation, training, and testing of vision-based systems designed for planetary exploration. Furthermore, the rapid development pace of Unreal Engine suggests the imminent introduction of even more advanced features.

Looking forward, we plan to continue analyzing these tools and further develop the rendering pipeline. As mentioned, a key aspect of our future work will involve comparing Unreal Engine 5 renders with a path-tracing benchmark such as Blender Cycles, to assess the impact of Nanite and Lumen approximations in

*https://quixel.com/megascans/

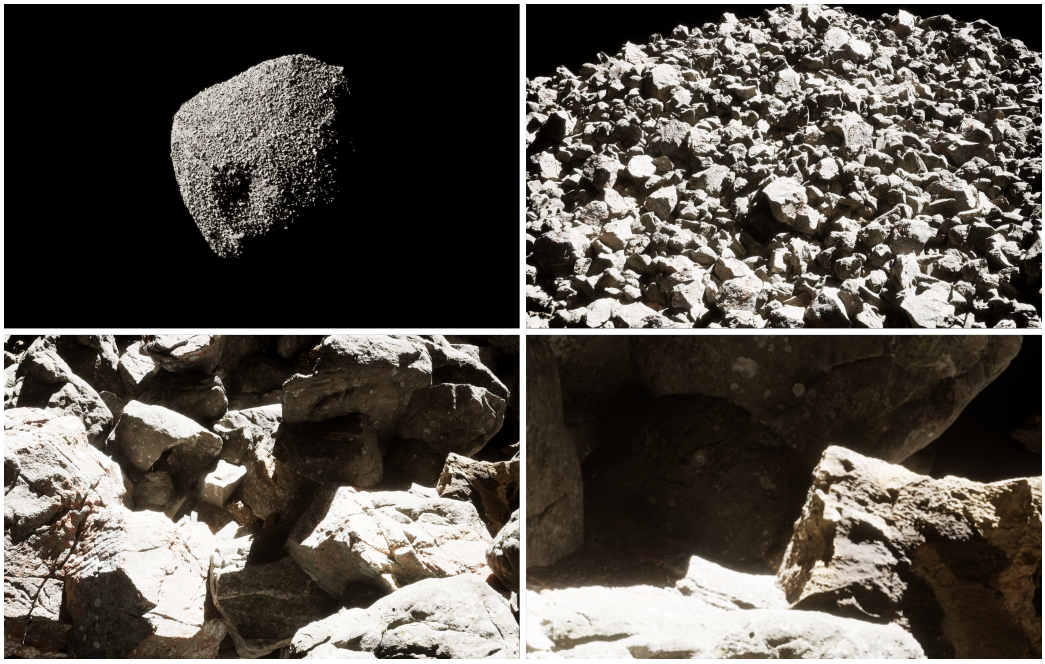


Figure 9: Screen captures of a synthetic rubble-pile asteroid, generated in UE5 using procedural-foliage tools, at varying camera distances. The scene is rendered in real time on a laptop, at multiple frames per second. Some of these frames contain at least billions of triangular surface elements. Starting from ten rock samples, the rock population is randomized in terms of size, orientation, and density. The rock instances are terrestrial items and are used solely for demonstration purposes.

the context of validation, training, and testing of vision-based systems. We also aim to enhance our camera modeling capabilities to better capture error sources and phenomena linked to real cameras, such as a realistic exposure time. Finally, we will refine our procedural-terrain generation pipelines, adjusting parameters and models to better replicate realistic shapes, size-frequency distributions, and densities of both craters and rocky features.

ACKNOWLEDGEMENTS

A portion of this research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. ©2023 California Institute of Technology. Government sponsorship acknowledged.

The authors would like to thank Dr. Benjamin Hockman from the Jet Propulsion Laboratory for the insightful exchanges and guidance through the course of this study.

REFERENCES

[1] D. Lauretta, S. Balram-Knutson, E. Beshore, W. Boynton, C. Drouet d'Aubigny, D. DellaGiustina, H. Enos, D. Golish, C. Hergenrother, E. Howell, *et al.*, "OSIRIS-REx: sample return from asteroid (101955) Bennu," *Space Science Reviews*, Vol. 212, No. 1, 2017, pp. 925–984.

[2] S.-i. Watanabe, Y. Tsuda, M. Yoshikawa, S. Tanaka, T. Saiki, and S. Nakazawa, "Hayabusa2 mission overview," *Space Science Reviews*, Vol. 208, No. 1, 2017, pp. 3–16.

[3] A. S. Rivkin, N. L. Chabot, A. M. Stickle, C. A. Thomas, D. C. Richardson, O. Barnouin, E. G. Fahnestock, C. M. Ernst, A. F. Cheng, S. Chesley, *et al.*, "The double asteroid redirection test (DART): Planetary defense investigations and requirements," *The Planetary Science Journal*, Vol. 2, No. 5, 2021, p. 173.

[4] G. Jones and C. Snodgrass, "Comet Interceptor: A proposed ESA Mission to a Dynamically New Comet.," *Geophysical Research Abstracts*, Vol. 21, 2019.

[5] R. D. Lorenz, E. P. Turtle, J. W. Barnes, M. G. Trainer, D. S. Adams, K. E. Hibbard, C. Z. Sheldon, K. Zacny, P. N. Peplowski, D. J. Lawrence, *et al.*, "Dragonfly: A rotorcraft lander concept for scientific exploration at Titan," *Johns Hopkins APL Technical Digest*, Vol. 34, No. 3, 2018, p. 14.

[6] C. Norman, C. Miller, R. Olds, C. Mario, E. Palmer, O. Barnouin, M. Daly, J. Weirich, J. Seabrook, C. Bennett, *et al.*, "Autonomous Navigation Performance Using Natural Feature Tracking during the OSIRIS-REx Touch-and-Go Sample Collection Event," *The Planetary Science Journal*, Vol. 3, No. 5, 2022, p. 101.

[7] A. E. Johnson, S. B. Aaron, H. Ansari, C. Bergh, H. Bourdu, J. Butler, J. Chang, R. Cheng, Y. Cheng, K. Clark, *et al.*, "Mars 2020 Lander Vision System Flight Performance," *AIAA SciTech 2022 Forum*, 2022, p. 1214.

[8] I. Martin, M. Dunstan, and M. S. Gestido, "Planetary surface image generation for testing future space missions with pangu," *2nd RPI Space Imaging Workshop*, Sensing, Estimation, and Automation Laboratory, 2019.

[9] J. Lebreton, R. Brochard, M. Baudry, G. Jonniaux, A. H. Salah, K. Kanani, M. L. Goff, A. Masson, N. Ollagnier, P. Panicucci, *et al.*, "Image simulation for space applications with the SurRender software," *arXiv preprint arXiv:2106.11322*, 2021.

[10] C. Aiazzi, A. Jain, A. Gaut, A. Young, and A. Elmquist, "IRIS: High-fidelity Perception Sensor Modeling for Closed-Loop Planetary Simulations," *AIAA SCITECH 2022 Forum*, 2022, p. 1433.

[11] M. Pajusalu, I. Iakubivskyi, G. J. Schwarzkopf, O. Knuuttila, T. Väisänen, M. Bührer, M. F. Palos, H. Teras, G. Le Bonhomme, J. Praks, *et al.*, "SISPO: space imaging simulator for proximity operations," *PloS one*, Vol. 17, No. 3, 2022, p. e0263882.

[12] J. D. Foley, *Computer graphics: principles and practice*, Vol. 12110. Addison-Wesley Professional, 1996.

[13] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[14] D. S. Immel, M. F. Cohen, and D. P. Greenberg, "A radiosity method for non-diffuse environments," *Acm Siggraph Computer Graphics*, Vol. 20, No. 4, 1986, pp. 133–142.

[15] J. T. Kajiya, "The rendering equation," *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 143–150.

[16] B. Hapke, *Theory of reflectance and emittance spectroscopy*. Cambridge university press, 2012.

[17] M. Oren and S. K. Nayar, "Generalization of Lambert's reflectance model," *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, 1994, pp. 239–246.

[18] H. Seeliger, "Zur photometrie des saturnringes," *Astronomische Nachrichten*, Vol. 109, 1884, p. 305.

[19] A. Kuzminykh, *Physically Based Real-Time Rendering of the Moon*. PhD thesis, Hochschule Hannover, 2021.

[20] L. Gritz, C. Stein, C. Kulla, and A. Conty, "Open shading language," *ACM SIGGRAPH 2010 Talks*, pp. 1–1, 2010.

[21] K. Perlin, "An image synthesizer," *ACM Siggraph Computer Graphics*, Vol. 19, No. 3, 1985, pp. 287–296.

[22] S. Worley, "A cellular texture basis function," *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 291–294.

[23] R. L. Cook, L. Carpenter, and E. Catmull, "The Reyes image rendering architecture," *ACM SIGGRAPH Computer Graphics*, Vol. 21, No. 4, 1987, pp. 95–102.

[24] K.-H. Glassmeier, H. Boehnhardt, D. Koschny, E. Kührt, and I. Richter, "The Rosetta mission: flying towards the origin of the solar system," *Space Science Reviews*, Vol. 128, No. 1, 2007, pp. 1–21.

[25] F. Preusker, F. Scholten, K.-D. Matz, T. Roatsch, S. Hviid, S. Mottola, J. Knollenberg, E. Kührt, M. Pajola, N. Oklay, *et al.*, "The global meter-level shape model of comet 67P/Churyumov-Gerasimenko," *Astronomy & Astrophysics*, Vol. 607, 2017, p. L1.

[26] B. J. Davidsson, B. J. Buratti, and M. D. Hicks, "Albedo variegation on Comet 67P/Churyumov–Gerasimenko," *Monthly Notices of the Royal Astronomical Society*, Vol. 516, No. 4, 2022, pp. 5125–5142.

[27] K. R. Castleman, *Digital image processing*. Prentice Hall Press, 1996.

[28] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE transactions on systems, man, and cybernetics*, Vol. 9, No. 1, 1979, pp. 62–66.

[29] H. Sato, M. Robinson, B. Hapke, B. Denevi, and A. Boyd, "Resolved Hapke parameter maps of the Moon," *Journal of Geophysical Research: Planets*, Vol. 119, No. 8, 2014, pp. 1775–1805.

[30] M. Pugliatti and F. Topputo, "DOORS: Dataset fOr bOuldeRs Segmentation. statistical properties and blender setup," *arXiv preprint arXiv:2210.16253*, 2022.

APPENDIX A: HAPKE MODEL

The modern version of the Hapke model, as published in 2012, is presented in Equation 7. Such a model is associated with nine free parameters: w, b, c, B_{C0}, h_C, B_{S0}, h_S, $\bar{\theta}_p$, ϕ, described below. The reader may find a full derivation in Hapke's seminal work.[16]

Let the incidence angle i and emission angles e be defined by

$$\cos(i) = \omega_i^T \cdot \mathbf{n} \tag{8}$$

and

$$\cos(e) = \omega_r^T \cdot \mathbf{n} \tag{9}$$

Let also g be the phase angle, defined by

$$\cos(g) = \omega_i^T \cdot \omega_r \tag{10}$$

In the following, the mathematical terms composing Equation 7 are provided and briefly described, using the notation from Sato et al.[29] or Kuzminykh.[19]

- w represents the average single-scattering albedo, defined as the ratio of scattered energy to extinct energy within a particle.
- $LS(i_e, e_e)$ symbolizes the Lommel-Seeliger function, which models the single-scattering component, i.e., the semi-infinite layer behavior, of regolith reflectance. It is expressed as:

  $$LS(i_e, e_e) = \frac{\cos(i_e)}{\cos(i_e) + \cos(e_e)} \tag{11}$$

  where i_e and e_e are the *effective* angle of incidence and angle of emission, respectively. Such angles account for the effect of surface roughness, whereas the standard i and e do not. Their definitions depends on the lighting geometry, as described in the subsequent paragraphs.

  Notice that $LS(i_e, e_e)$ is parametrized differently than in the Lommel-Seeliger BRDF (Equation 6), which makes use of the regular incidence and emission angles, i and e, respectively.
- $p(g)$ denotes the Henyey-Greenstein double-lobed particle phase functions, a prevalent model that represents the scattering distribution from a single particle. This is given by

  $$p(g) = \frac{1+c}{2}\frac{1-b^2}{(1-2b\cos(g)+b^2)^{3/2}} + \frac{1-c}{2}\frac{1-b^2}{(1+2b\cos(g)+b^2)^{3/2}} \tag{12}$$

  where b is the shape-controlling parameter, controlling the lobes' shape and amplitude, where $0 \leqslant b \leqslant 1$. c is the relative strength of backward and forward lobes, where $-1 \leqslant c \leqslant 1$.
- $B_S(g)$ is the Shadow Hiding Opposition Effect (SHOE) function, expressed as

  $$B_S(g) = 1/[1 + \tan(g/2)/h_S] \tag{13}$$

  in which h_S is the angular width of the SHOE. The SHOE has been further described in the main body of this manuscript.
- B_{S0} is the SHOE amplitude coefficient.

- $M(i_e, e_e)$ is the Isotropic Multiple Scattering Approximation (IMSA), which models the effect of multiple scatters throughout the regolith medium. IMSA is given by:

$$M(i_e, e_e) = H\left(\frac{\cos(i_e)}{K}, w\right) H\left(\frac{\cos(e_e)}{K}, w\right) - 1 \tag{14}$$

where K is the porosity factor, representing how porous the regolith medium is, computed as

$$K = \frac{-\ln(1 - 1.209\phi^{2/3})}{1.209\phi^{2/3}} \tag{15}$$

where $\phi(= 1 - \text{porosity})$ is the filling factor.

$H(x, w)$ is the Ambartsumian-Chandrasekhar H function, also known as the H-function. It is used to characterize the radiative transfer of light through the regolith medium. The H-function can be approximated as

$$H(x, w) \simeq \left\{1 - wx\left[r_0 + \frac{1 - 2r_0 x}{2}\ln\left(\frac{1+x}{x}\right)\right]\right\}^{-1} \tag{16}$$

where r_0 is the diffusive reflectance:

$$r_0 = \frac{1 - \sqrt{1-w}}{1 + \sqrt{1-w}} \tag{17}$$

- $B_C(g)$ is the Coherent Backscatter Opposition Effect (CBOE) function, given by

$$B_C(g) = \frac{1 + \dfrac{1 - \exp[-\tan(g/2)/h_C]}{\tan(g/2)/h_C}}{2[1 + \tan(g/2)/h_C]^2} \tag{18}$$

where h_C is the angular width of the CBOE.

- B_{C0} is the CBOE amplitude coefficient.

- $S(i, e, \psi)$ is the shadowing (or surface roughness) function. This is used to model shadowing within the regolith, at a microscopic level, and how it affects the overall reflectance. $S(i, e, \psi)$ depends on $\bar{\theta}_p$, which is the average angle between the macroscopic surface normal and the normal of a microscopic surface facet. Effectively, $\bar{\theta}_p$ controls the surface roughness.

  $S(i, e, \psi)$ depends on the lighting geometry. For $i \leqslant e$:

$$S(i, e, \psi) = \frac{\mu_e}{\eta(e)} \frac{\mu_0}{\eta(i)} \frac{\chi(\bar{\theta}_p)}{1 - f(\psi) + f(\psi)\chi(\bar{\theta}_p)[\mu_0/\eta(i)]} \tag{19}$$

where

$$\mu_{0e} = \cos(i_e) = \chi(\bar{\theta}_p)\left[\cos(i) + \sin(i)\tan(\bar{\theta}_p)\frac{\cos(\psi)E_2(e) + \sin^2(\psi/2)E_2(i)}{2 - E_1(e) - (\psi/\pi)E_1(i)}\right] \tag{20}$$

and

$$\mu_e = \cos(e_e) = \chi(\bar{\theta}_p)\left[\cos(e) + \sin(e)\tan(\bar{\theta}_p)\frac{E_2(e) - \sin^2(\psi/2)E_2(i)}{2 - E_1(e) - (\psi/\pi)E_1(i)}\right] \tag{21}$$

For $e \leqslant i$:

$$S(i,e,\psi) = \frac{\mu_e}{\eta(e)}\frac{\mu_0}{\eta(i)}\frac{\chi(\bar{\theta}_p)}{1-f(\psi)+f(\psi)\chi(\bar{\theta}_p)[\mu/\eta(e)]} \tag{22}$$

where

$$\mu_{0e} = \cos(i_e) = \chi(\bar{\theta}_p)\left[\cos(i) + \sin(i)\tan(\bar{\theta}_p)\frac{E_2(i) - \sin^2(\psi/2)E_2(e)}{2 - E_1(i) - (\psi/\pi)E_1(e)}\right] \tag{23}$$

and

$$\mu_e = \cos(e_e) = \chi(\bar{\theta}_p)\left[\cos(e) + \sin(e)\tan(\bar{\theta}_p)\frac{\cos(\psi)E_2(i) + \sin^2(\psi/2)E_2(e)}{2 - E_1(i) - (\psi/\pi)E_1(e)}\right] \tag{24}$$

$\chi(\bar{\theta}_p)$ is given by

$$\chi(\bar{\theta}_p) = 1/[1 + \pi\tan^2(\bar{\theta}_p)]^{1/2} \tag{25}$$

and $\eta(y)$, for $y = \{i, e\}$, is given by

$$\eta(y) = \chi(\bar{\theta}_p)\left[\cos(y) + \sin(y)\tan(\bar{\theta}_p)\frac{E_2(y)}{2 - E_1(y)}\right] \tag{26}$$

In this equation, $E_1(y)$ and $E_2(y)$, for $y = \{i, e\}$, are

$$E_1(y) = \exp\left[-\frac{2}{\pi}\cot(\bar{\theta}_p)\cot(y)\right] \tag{27}$$

$$E_2(y) = \exp\left[-\frac{1}{\pi}\cot^2(\bar{\theta}_p)\cot^2(y)\right] \tag{28}$$

Lastly, the term $f(\psi)$ in $S(i,e,\psi)$ is computed as

$$f(\psi) = \exp\left[-2\tan\left(\frac{\psi}{2}\right)\right] \tag{29}$$

where ψ is the azimuth angle, such that

$$\cos(g) = \cos(e)\cos(i) + \sin(e)\sin(i)\cos(\psi) \tag{30}$$

hence, it can be written as

$$\psi = \arccos\left[\frac{\cos(g) - \cos(e)\cos(i)}{\sin(e)\sin(i)}\right] \tag{31}$$

APPENDIX B: BLENDER CYCLES NODE SYSTEM FOR PROCEDURAL ASTEROIDS

The Blender Cycles node system used to generate the fully procedural asteroids, as shown in Figure 8, is depicted in Figure 10.

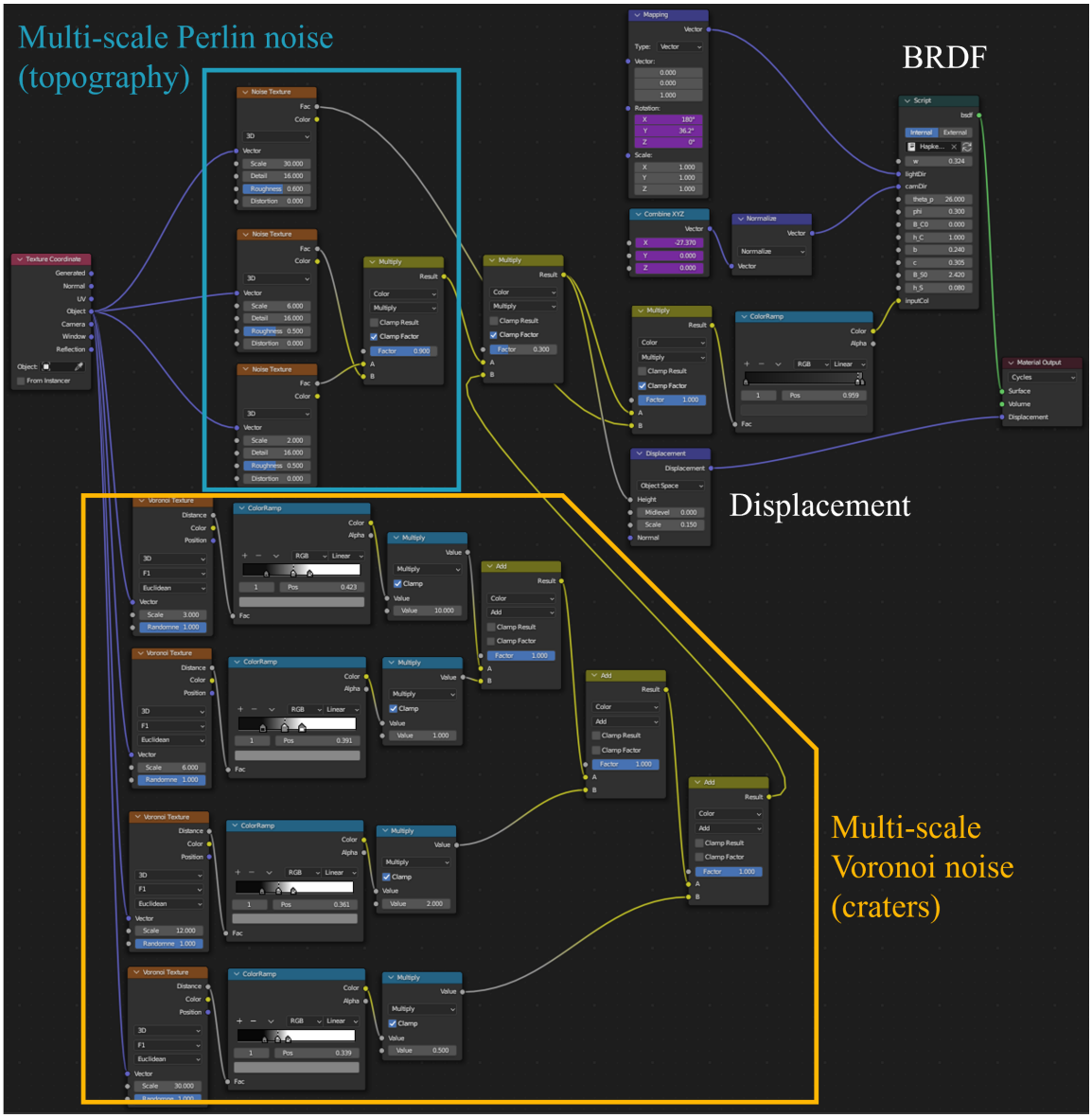


Figure 10: Screen capture of the node system in Blender Cycles used for generating the fully procedural asteroids seen in Figure 8. This system uses multi-scale Perlin noise (top) to create terrain topography and macroscopic displacement of the original spherical shape. It also uses multi-scale Voronoi noise (bottom) to generate a crater distribution. Both types of noise functions serve as albedo and elevation maps, and are passed to the BRDF nodes (top-right) and the displacement nodes (bottom-right) for runtime evaluation.