Dspace: Real-time 3D Visualization System for Spacecraft Dynamics Simulation

Marc I. Pomerantz, Abhinandan Jain, Steven Myint

Jet Propulsion Laboratory, California Institute of Technology

4800 Oak Grove Drive, Pasadena, CA 91109

Marc.I.Pomerantz@jpl.nasa.gov

Abstract*—The multi-mission Dshell++ simulation framework has formed the basis for high-performance, physics-based simulations for a large variety of space mission simulations including cruise and orbiter spacecraft, Entry, Descent, Landing (EDL) missions as well as for planetary surface rovers. The Dspace interactive, reusable 3D visualization system has been developed to support the diverse visualization needs of such complex real-time simulations

I. Introduction

The Dshell++ [1] physics-based simulation framework has been used to develop simulations in support of a variety of NASA space mission domains and is the basis for high-performance, reconfigurable, simulation software.

The Dspace 3D visualization system has been developed to support the Dshell++ simulation software suite as a run-time configurable solution designed to: provide accurate simulation visualization; minimize simulation development time and costs; meet the real-time simulation performance speeds; support graphics related computations for relevant simulation models. Close interaction between a simulation and visualization is important to maintain visual accuracy. Dspace supports this close interaction by providing an API that allows users to continuously update object position and orientation and to display: planetary bodies at their correct positions; object trails and shadows; simulation generated engineering data such as vehicle telemetry or false color terrain height or normal maps; goal or vehicle tilt markers.

While using many of the software concepts and techniques commonly used in third-party visualization software applications or libraries such as Blender [2], Ogre [3] and Coin3D [4], Dspace provides value-added capabilities for organizing viewports and scene graph elements such as geometry, lighting and cameras, which is essential when performing *Selective Rendering* as described below. Additionally, Dspace supports real-time, high-performance, graphics processing unit (GPU) based rendering of very large terrain data sets, vehicle shadows and wheel tracks, visualization based engineering analysis and rendering synthetic imagery for camera modeling, and an easy to use API that allows simulation developers to construct custom, reusable, visualizations while ensuring kinematic accuracy between simulation and visualization.

Supported Dspace visualization domains include: spacecraft trajectory visualization containing planetary bodies, star maps and spacecraft; user defined rover vehicle visualizations on high-resolution, real terrain with wheel tracks, shadows and multiple vehicle cameras such as ROAMS [5]; Entry, Decent and Landing visualizations depicting from-orbit to landing simulations with various mission events including heat shield separation, parachute deployment, thruster firing and landing such as DSENDS[6]; rover vehicles in an urban setting including buildings, streets and other vehicles; visualization of *Command and Control* simulations that include orbital communication assets and theatre-level maps and information gathering assets.

Dspace contains a rich feature set that includes: synthetic camera modeling, high-performance terrain visualization, real-time shadows and vehicle wheel tracks, the accurate representation of spacecraft vehicle kinematics, multiple scene viewpoints, image-based horizon detection and line of sight modeling, and an object-oriented open architecture developed in C++ and Python [7].

This paper will discuss Dspace's requirements and design, application interface, and integration with the Dshell++ simulation architecture along details of its various features and extensions.

II. Requirements and Software Design

A. Requirements

Dshell++ has been designed as a multi-mission physics-based simulation framework to support space system simulation needs across multiple domains including cruise vehicles, Entry, Descent and Landing, planetary rovers, aerobots and orbiting spacecraft. Based on these broad simulation needs, we identified a core set of visualization and 3D graphics needs to support such high-fidelity simulations. To reduce simulation cost and development time, we adopted a multi-mission, data-driven, run-time configurable system design that requires no changes to visualization source code from one simulation to another.

Additional capabilities such as 30 frame per second rendering performance, realistic terrain rendering, synthetic image synthesis for vehicle camera modeling, real time vehicle shadows and wheel tracks (as shown in Fig. 1), managing physical objects such as terrain and vehicles and artificial objects such as goal markers, waypoints markers or coordinate axes were included in the requirements set.

Because 3D visualization is computationally expensive, it was deemed advantageous to distribute the simulation/visualization computation load across multiple cores by running in a client/server mode either on one workstation or across multiple workstations. If the user so desires, Dspace can also be used as a linked, run-time library in which the simulation and Dspace run as a single process.

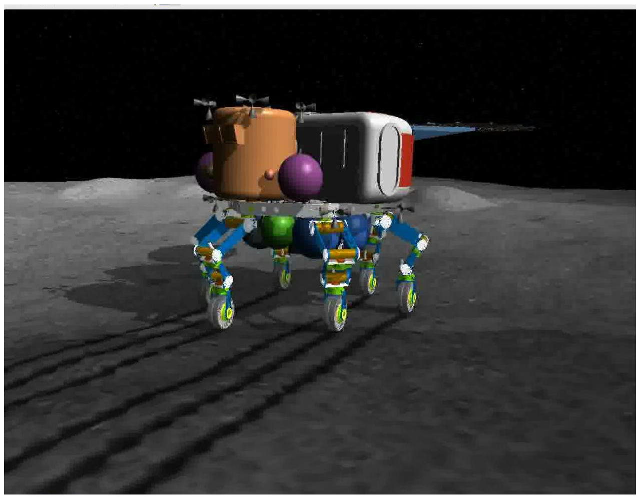
Figure 1. Dspace rendering with articulated vehicle, real-time shadows and wheel tracks.

B. *Software Design*

Dspace has been designed as an object-oriented, open architecture, scene graph-based system. Originally built on top of the SGI OpenInventor [8] library, Dspace now uses the Coin3D library and OpenGL [9], Dspace's core classes manage viewports, cameras, scene graph fragments, graphics objects (geometry, textures, transformations, etc.), reference frames and scene vantage information and are typically derived from Coin3D base classes. An ancillary library, DspaceTerrain has been developed to manage and render terrain and planetary body data. Fig. 2 depicts the Dspace software organization.

Dspace contains a core set of classes for creating, organizing, managing and displaying scene graphs. Viewports provide scene graph traversal and rendering mechanism, along with event handling such as mouse or keyboard interaction; GraphicsObjects import scene graphs as contained in the Virtual Reality Modeling Language (VRML) disk files [10] and provide routines for changing scene graph states, such as visibility and material properties;

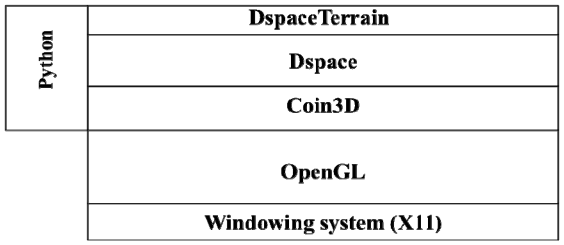

Figure 2. Dspace software modules.

SceneFragments contain a collection of scene graphs as provided by multiple GraphicsObjects and can be used to segregate physical from ornamental scene graphs; SceneFragmentManagers combine multiple SceneFragments and provides routines for managing lighting to ensure that light objects are placed at the top of the scene graph hierarchy for complete scene graph illumination; SceneVantages each combine a SceneFragmentManager with one or more Camera instances, providing a view or vantage of the scene at render time.

Multiple Viewport instances are supported, each with a unique Camera instance as provided by the Viewport's contained SceneVantage, and can either share scene graphs or can contain unique scene graph information. The sharing of scene graphs allow multiple Viewports to display different views of the same scene.

In addition, Dspace supports a notion of selective rendering, where users can group GraphicsObjects based on type or other specified taxonomy. SceneFragments serve as the container class for specific groups and when required during rendering, individual SceneFragments can be "hidden" which cause their contained GraphicsObjects not to be rendered. For example, when performing camera modeling, Viewports representing the left and right stereo camera views display identical scene graph information, but only render GraphicsObjects that represent real-world physical objects, while a third Viewport might display a global view of the scene, including physical GraphicsObjects depicting terrain and vehicles and ornamental GraphicsObjects representing goal markers or coordinate axes.

1) *GraphicsObject*

Each set of geometric primitives, such as triangles, points, lines or text, to be displayed by Dspace, must reside in a scene graph contained in a GraphicsObject (see below). These scene graph elements are typically imported VRML files, but can be made up of primitives, created at runtime, such as cones, spheres, cubes, transformations, materials, textures or other supported Coin3D nodes.

The GraphicsObject class imports scene geometry in VRML format, supports multiple levels-of-detail on a per-Graphics Object basis and is used to represent both physical objects (vehicles/terrain/planetary bodies) and ornamental objects (goal markers, coordinate axes, 3D text).

An individual GraphicsObject's scene graph fragment is combined with other GraphicsObject's scene graph fragments in the SceneFragment to form a single more complex scene graph. A GraphicsObject instance *must* be added to a SceneFragment instance in order to be processed during scene graph render traversals.

The GraphicsObject class is also the base class for any user written, specialized class that requires scene graph traversal at render time. The DspaceTerrain library specializes this class for rendering terrain geometry.

2) SceneFragment

Individual scene graph fragments, contained in GraphicsObject instances, are collected by the SceneFragment class and can be organized based on type or group information. Physical and ornamental GraphicsObects can be segregated for controlled display during synthetic camera imaging or traversal planning. For example, when generating synthetic camera imagery, SceneFragments containing ornamental GraphicsObjects can be directed to "hide" themselves during rendering, thereby causing final render images to contain only physical objects such as terrain or rovers. Through the use of this mechanism, SceneFragments can be shown or hidden as directed by the simulation as a way to selectively render collections of GraphicsObjects at render time.

3) SceneFragmentManager

Prior to the rendering cycle traversal, scene graphs are collected into a more complex scene graph and contained in the SceneFragment. Multiple SceneFragments are then passed to the SceneFragmentManager, which makes the contained scene graph available to the SceneVantage along with functions for SceneFragment management such as the addition of new SceneFragments and the deletion of contained SceneFragments.

4) SceneVantage

SceneFragmentManagers contain scene graph fragments and Viewports manage scene graph traversal at render time, but something must connect the scene graph to the Viewport. This connecting code resides in the SceneVantage class. The SceneVantage class contains a scene graph as provided by a SceneFragmentManager (the "Scene") and a camera or set of cameras (the "Vantage"). When directed by the Viewport at render time, the SceneVantage instances traverse their entire contained scene graphs and viewing the scene geometry using the selected Camera. Fig. 3 shows how multiple SceneFragments and SceneFragmentManagers are utilized by one or more viewports.

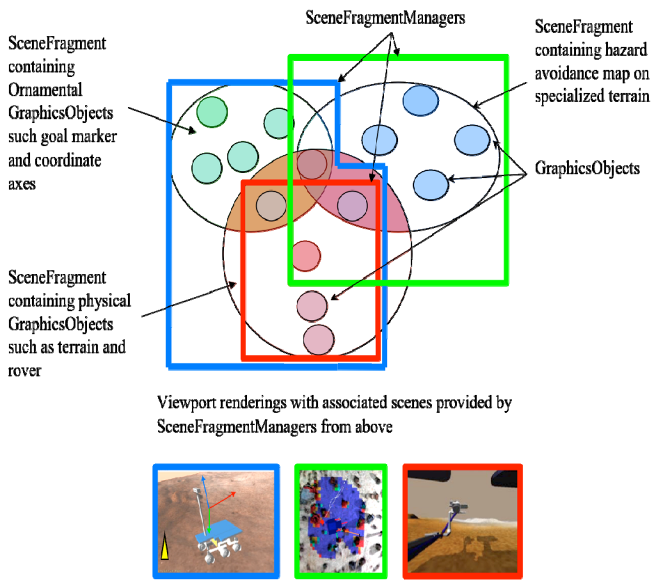
SceneFragmentManagers
SceneFragment containing Ornamental GraphicsObjects such goal marker and coordinate axes
SceneFragment containing hazard avoidance map on specialized terrain
GraphicsObjects
SceneFragment containing physical GraphicsObjects such as terrain and rover
Viewport renderings with associated scenes provided by SceneFragmentManagers from above

Figure 3. GraphicsObjects, SceneFragmentManagers and Viewport organization allow segregated rendering.

SceneVantage instances can contain multiple Cameras but only one selected Camera may be active at any given time. Scene graph traversal is typical for a *retained mode* renderer, but if shadows are activated, a two pass rendering process based on the Z-buffer shadowing [11] algorithm is performed, with final shadow determination performed in a fragment shader**.**

5) Frames

Frame classes provide the backbone or skeletal system for the entire scene graph and provide a mechanism for articulating the components of multi-body objects, such as roving vehicles.

Frame instances provide four important functions at runtime. First, Frames provide an attachment point for geometry as shown in Fig. 4. For example, a rover chassis frame may have the geometry for the chassis, solar and camera mast all attached. Second, Frames can be attached to other Frames, thereby forming the parent-child hierarchical scene graph. Third, transformation information as received from the Dshell++ simulation is stored in Frame instances. When transformations are applied to a Frame, all attached geometry are translated, rotated or scaled accordingly. Fourth, when specified as Physical or Ornamental, attached GraphicsObjects with their contained geometry are provided to the appropriate SceneFragment by the Frame, thereby segregating physical from ornamental objects at render time. This is important when performing selective rendering, as discussed later and shown in Fig. 7, for synthetic camera

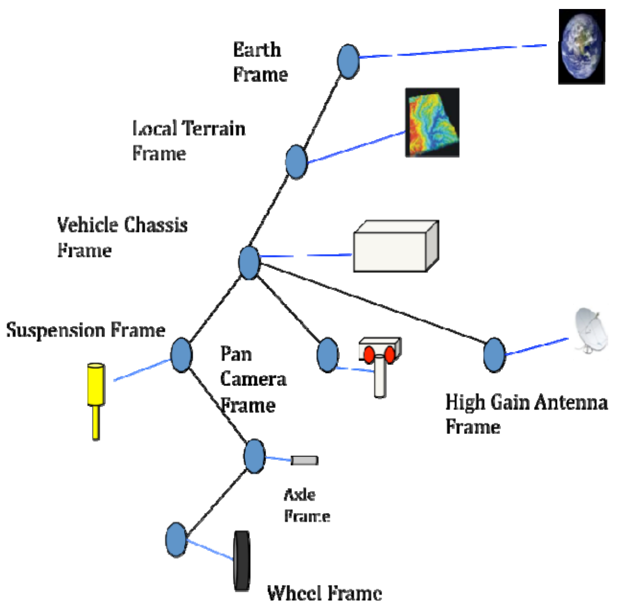

Figure 4. Dspace frames provide scene graph backbone.

imagery where only physical objects (terrain, vehicles, planets) are to be rendered.

6) *Cameras*

Dspace supports three different types of Coin3D/OpenGL cameras: Perspective, Orthographic and Frustum. These cameras can be arbitrarily placed and oriented under user control, either through mouse interaction, or through simulation control via the Dspace API. Typical OpenGL camera operations are supported such as position, orientation, frustum settings and clipping plane values. The Dspace camera classes are derived from the Coin3D base classes and provide convenience routines for positioning and pointing cameras when following designated simulation objects. This mechanism is used to provide "chase" views of spacecraft or rover vehicles.

7) *Viewports*

The Dspace Viewport class is a container for a single Scene Vantage instance and also provides routines used for performing scene graph traversal and rendering, user interaction and event handling (mouse or keyboard), interaction with the native windowing system (typically X11) and displaying the final rendered 3D scene images along with any simulation provided tabular engineering information. Users can create, resize, hide and place Viewports, control camera position and pointing and select 3D scene items via mouse interaction. Dspace allows users to create an arbitrary number of Viewports, only limited by workstation resources.

III. DSHELL++ SIMULATION-DSPACE INTEGRATION

During a Dshell++ simulation run, the initialization process is responsible for registering simulation objects with Dspace in the correct scene graph hierarchy. Dshell++ does this by directing Dspace to create Dspace Frame instances, which in essence are proxy classes for Dshell++'s own Frame instances. Every Dshell++ simulation object, from planetary bodies to individual parts of a multi-body system, such as a spacecraft or roving vehicle, contain a Dshell++ Frame.

A. Synchronizing the simulation and the visualization

As simulation object status is updated during simulation execution, Dspace's scene must be updated accordingly. The Dshell++ simulation manages this efficiently at run-time using watch handlers that trigger only when the state of a simulation object changes. For articulated rover vehicle simulations, joint motions for vehicle components, such as suspension systems or camera mast parts, trigger simulation callbacks that send frame information (position and orientation) to the corresponding Dspace Frame. Updating Dspace Frame information happens continuously in real time and keeps the Dspace visualization state in sync with the simulation state at all times while minimizing the amount of update traffic flowing from the simulation to Dspace.

B. Kinematic Accuracy

Because users rely significantly on the Dspace visualization to interpret the simulation state, it is essential that the visualization model correspond accurately to the simulation model geometry and kinematics.

Keeping graphics models consistent with simulation models can be a challenging and expensive proposition as the vehicle structure and design evolves, especially during the early phases of projects. The described one-to-one correspondence between Dshell++ Frames and Dspace proxy Frames mitigates this potential problem. This approach essentially consists of Dshell++ auto-generating a skeleton scene graph that is derived from, and faithful to the underlying physical model within the simulation. The 3D parts geometry of the vehicle, such as wheels, masts, arms, etc., are attached to this skeleton to ensure that the simulation and visualization models maintain close correspondence to each other. In this way, Dshell++ essentially directs Dspace to construct the 3D geometry of the scene based on accurate simulation data each time that the simulation and Dspace run as in Fig. 5. If a vehicle component, specification or characteristic changes from run to run, only simulation changes are needed, no changes to Dspace are required.

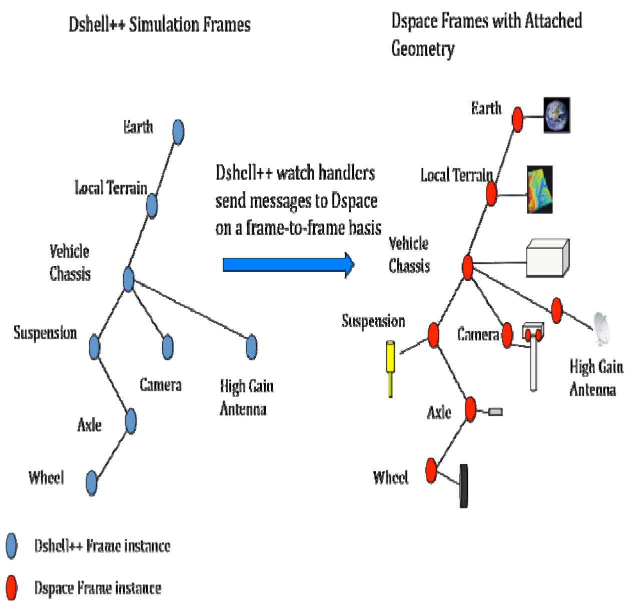
Dshell++ Simulation Frames
Dspace Frames with Attached Geometry
Earth
Local Terrain
Vehicle Chassis
Suspension
Camera
High Gain Antenna
Axle
Wheel
Dshell++ watch handlers send messages to Dspace on a frame-to-frame basis
Dshell++ Frame instance
Dspace Frame instance

Figure 5. Dshell++ to Dspace Frame connections ensure kinematic accuracy.

IV. Dspace Feature Set

A. High Performance Rendering

To support the increasing complexity of space mission simulations, Dspace combines fixed functionality OpenGL with OpenGL Shading Language [12] (GLSL) shader code to perform rendering operations that would be prohibitively slow with fixed functionality OpenGL alone. GLSL shader code essentially re-programs the graphics processing unit (GPU), on the workstation's graphics card, to perform custom operations that may not be supported using fixed-functionality OpenGL, and executes that shader code concurrently on the GPU's multiple processing cores. As described above, vehicle shadows, performed in a two pass rendering process and computed in a fragment shader can be rendered, while maintaining 30+ fps frame rates. Fig. 6 shows a Dspace rendering with rover shadows. Vehicle wheel tracks, terrain characteristic information such as height and normal maps, multi-texturing and the use of the Perlin [13] noise algorithm to enhance terrain realism are also performed in OpenGL Shading Language vertex and fragment shaders.

B. User Interaction

Users can interact with the Dspace rendered scene and can modify camera viewpoint information via mouse interaction. In addition, Dspace can be used to acquire data about the 3D scene. For example, by picking a point on the terrain, inertial and body fixed coordinates are returned that can be used by the Dshell++ simulation to set vehicle goal information or vehicle placement.

1) Rich User API

When running in single process (local) mode, with Dspace linked as a library, all of Dspace's provided core classes, as described above, can be invoked via C++ or Python. We use SWIG [14] to generate the Python to C++ interface classes.

When running in distributed (server) mode, with Dspace executing as a separate process from the simulation, we provide Python proxy classes for the core Dspace classes. The Python proxy classes send Python commands to Dspace via Python sockets. Because Dspace is a library of Python wrapped C++ classes, the user is free to create a Dspace application using either a C++ *main* routine or a Python script. When running as a Python application, users can access Dspace class instances and invoke class member functions.

When running in either local or server modes, users can custom-craft the look and feel of their Dspace visualization.

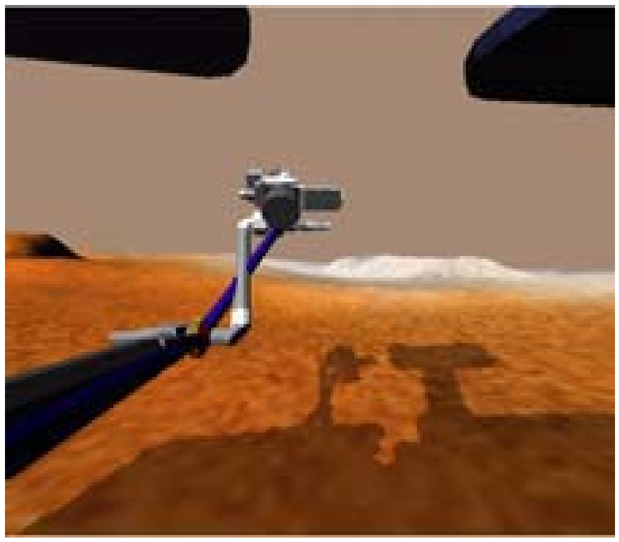

Figure 6. Rover camera view with real-time shadows

2) Selective Rendering

Rendered scenes typically contain "real" simulation objects together with additional "annotation" objects that can help the user interpret the state of the simulation. The Dspace framework provides ways to arbitrarily organize and group simulation objects by such properties to allow run-time selection of the visualization content by selective rendering of the groups. An important use case for selective rendering is that of rover camera modeling as described above, where only physical-based objects such as terrain, planetary bodies or rover components, should be seen in the final rendered image, and ornamental objects such as goal markers, viewing frustum pyramids or coordinate axes are to be excluded. Fig 7 depicts a typical scene graph backbone created through the use of Frame instances, with each Frame instance referencing attached geometry, with the attached geometry assigned to either physical or ornamental SceneFragments, for possible later use when performing

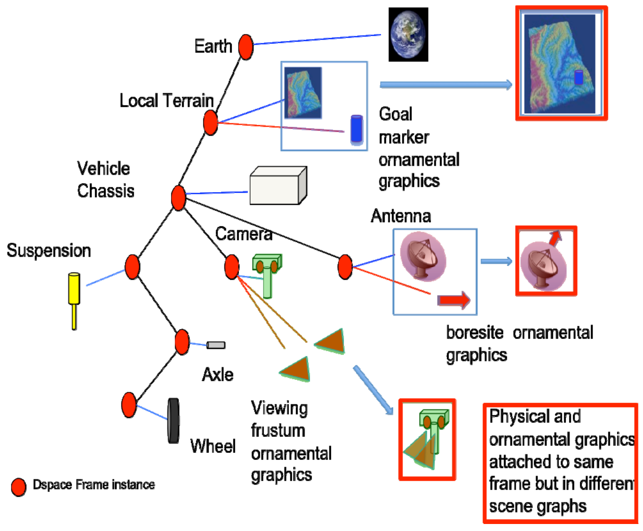
Earth
Local Terrain
Goal marker ornamental graphics
Vehicle Chassis
Antenna
Camera
Suspension
boresite ornamental graphics
Axle
Viewing frustum ornamental graphics
Wheel
Physical and ornamental graphics attached to same frame but in different scene graphs
Dspace Frame instance

Figure 7. Dspace Frame instances with attached ornamental geometry in ornamental SceneFragment (shown in red) and physical geometry in physical SceneFragment (shown in blue) allow for *Selective Rendering* as required by Camera Modeling.

selective rendering. The Frame attached geometry is assigned to a specific SceneFragment by the simulation at initialization time, using geometry type information provided by the user.

C. *Movies and Keyframe Camera Control*

Dspace supports the generation of movies of any given simulation run. Individual simulation frame renderings are combined to form an animation, with key frame controlled, smooth camera flight path supported. When a user wishes to create a movie, a simulation run is performed and a simulation log file of all Dspace commands can be saved to a file. A Python script called DspacePlayback.py reads that log file and segregates the log data into simulation frames. Because the log file contains every Python command that the simulation sent to Dspace, including the commands used to start Dspace, the DspacePlayback.py script can start up Dspace and replicate the entire simulation run in stand-alone mode, with no simulation in the loop.

The user can then change and save the camera positions and orientations, by mousing in a Dspace Viewport, for user selected frames over the entire frame set from the loaded log file. These selected frames are the camera key frames for the movie. Camera information for all other non-key frame frames is computed by interpolating from one key frame to the next. This method allows the user to create smooth camera flight paths for the movie without the need for a simulation in the loop.

An additional and often used movie feature is the ability to make many duplicates of any given frame as a way to freeze simulation time during a movie, but still provide camera key frame information over those duplicated frames. The result of this frame duplication is again smooth camera flight paths during a movie, but for those duplicated frames, simulation time does not advance.

V. Engineering Analysis

Dspace has been used not only to render scenes during a simulation, but to also generate engineering data to be used by the simulation. By taking advantage of the scene rendering capability in Dspace and by utilizing hardware acceleration provided by modern graphics cards, the operations described below can be executed quickly and are general enough to be used in a variety of simulation scenarios. The Lunar Surface Operations System [15] (LSOS) project simulation makes extensive use of the Dspace features described below

A. *Camera Modeling*

Dspace also supports modeling that is especially suited to graphics hardware acceleration. An important example of this is the generation of synthetic imagery for real-time camera simulation, to close the loop with machine vision algorithms and software. Simulated camera images are required to accurately capture the camera optics (including non-idealities) as determined by camera calibration parameters. The Dshell++ camera models [16] simulate radial and fish-eye distortion in camera lenses and make use of Dspace's offscreen rendering capability, as part of the camera modeling process, to achieve real-time performance when operating with the simulation in a closed-loop mode.

When performing camera modeling, the simulation makes a request to Dspace to render a camera view using the provided camera position and pointing information along with final image resolution. Dspace renders the scene and then returns the image back to the simulation for additional warping, as shown in Fig. 8, to apply the camera non-idealities, such as lens distortion. To avoid image size restrictions based on graphics hardware frame buffer size, Dspace renders camera model requests off-screen and can render images in excess of 4000x4000 pixels. Range map information can also be extracted during the camera modeling process.

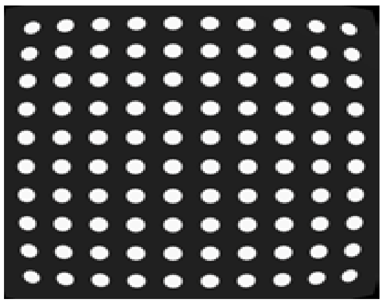

Figure 8. Dspace rendering of simulated calibration target board with simulated radial distortion applied.

B. *Line of Sight Computation*

Performing a line of sight computation can be computationally expensive when terrain and other vehicles or habitats can possibly obstruct the view from one viewer to another.

For a lunar rover simulation, line of sight communication is one of a variety of possible method for communication between a lunar vehicle and base station. Using the hardware-accelerated rendering in Dspace, we can perform these computations quickly and return results back to the simulation.

During simulation setup, Dshell++ sensors, simulating communication antennas, are placed at specified locations on the vehicle or habitat. In addition, specifically colored ornamental geometry is placed at the same location and is enabled during line of sight rendering. Dspace then renders the scene from the point of view of one vehicle's sensor while looking at the other vehicle's sensor.

If the specified sensor color is detected at the center of the rendered image, point-to-point line of sight has been achieved and is reported to the simulation. If the specified color is not detected, then it is assumed that a terrain feature, vehicle, or habitat component has obscured the view as in Fig. 9, and a failed line of sight is reported to the simulation.

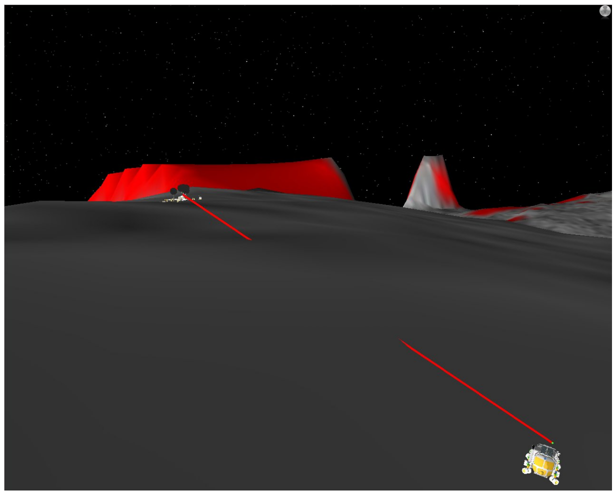
Figure 9. Terrain obstructed line of sight from lunar vehicle to habitat.

C. *Power Analysis*

At the Lunar South Pole region under examination by the LSOS project, the Sun stays close to or below the horizon for weeks at a time. In this environment, it is important to understand how much sunlight will fall on a habitat or vehicle mounted solar array, as in Fig. 10, and to compute the possible power generation.

Dspace uses an image based rendering technique, similar to the Line of Sight computation, to compute the percentage of solar array area illuminated by sunlight for a given point in time. Using planetary ephemeris, lunar topography and a size characterization of the solar array to be examined, Dspace renders from a point of view along the Sun vector, at a distance of approximately 200km and with a very narrow camera field of view. The 200km distance was chosen to ensure that the camera was far enough away from the solar array and lunar surface, to include all possible terrain features that might obscure sunlight.

At the completion of the Sun vector rendering, the rendered image buffer is examined and the total number of pixels representing solar array geometry is returned to the simulation where the total area of sunlight illuminated solar array is computed.

Dspace can detect pixels representing solar array geometry using a technique whereby the solar array elements are specifically colored, similar to the Line of Sight sensors. During the Power Analysis rendering pass, the normal solar array colors are replaced by the specified color and then restored at the completion of the Power Analysis rendering pass.

The simulation may request this Dspace computation many times over the course of a simulation run as the Sun

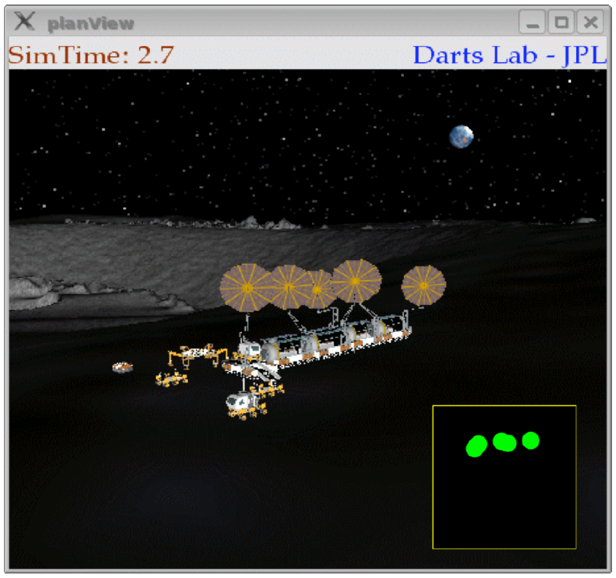

Figure 10. LSOS simulation of a potential lunar habitat system from NASA LaRC with inset Sun vector power analysis rendering.

moves in the lunar sky. The results of these requests are used to create a power profile over the course of lunar months or years.

D. *Horizon Detection*

To determine local terrain features surrounding a lunar habitat, a horizon scan capability has been added to Dspace. During a horizon scan, the simulation sends viewer location information to Dspace and requests that Dspace perform a full 360 degrees scan of the area around the viewer. This is done by rendering four different views, each with a 90 degree field of view and with the camera pointing vector rotated about the up vector (Z axis) 90 degrees per rendered image. The result is four rendered image buffers in memory. Dspace can then scan each rendered image and detect pixels lying exactly on the horizon. The inertial coordinate of each detected horizon pixel can be computed either by using the OpenGL gluUnProject function or by shooting a ray from the viewer to the horizon pixel.

The set of detected horizon pixels, along with their inertial coordinates, camera location and pointing information, is returned to the simulation for further processing.

VI. Terrain Rendering Using DspaceTerrain

Terrain data is typically the largest data set we use and render during a simulation run and includes geometry and other ancillary products such as textures and normal maps and may also contain soil and thermal properties and geo-reference information.

To manage this data, a specialized Dspace extension library called DspaceTerrain, has been developed to access terrain information from the SimScape [17] terrain modeling library, as either gridded digital elevation map (DEM) or latitude/longitude referenced gridded planetary data, and promote that data to scene graph information that can be traversed and rendered.

Simple terrain data can be exported as triangle strips for fast rendering, but larger, more complex data sets can be rendered using an implementation of the GPU Geometry Clipmap [18] algorithm. During scene graph traversal, GLSL shaders use a series of fixed meshes of varying resolutions and continually modify the height values of those mesh vertices based on the height values of the provided terrain data, with the highest resolution meshes closest to the viewer location and lower resolution meshes farther from the viewer, thereby minimizing the number of polygons to be rendered at each redraw cycle. The Clipmap shaders also perform multi-texturing with noise added if requested; compute and display terrain height and normal information; draw rover wheel tracks in the fragment shader when rover wheel position is provided; and can paint ornamental overlays onto the terrain such as latitude/longitude lines or false color depiction of the terrain as in Fig. 11.

Terrain grouping is also a supported feature whereby DspaceTerrain can combine reference or context planetary geometry with local, high-resolution terrain geometry. An example would be the combining and rendering of the entire Mars MOLA [19] terrain with a high-resolution Victoria Crater terrain patch [20].

VII. Conclusions

Dspace is currently used by a variety of NASA/JPL projects and provides a re-usable, cost-effective, run-time configurable solution for spacecraft and roving vehicle simulations that require interactive, high-performance, 3D visualization and targeted engineering analysis.

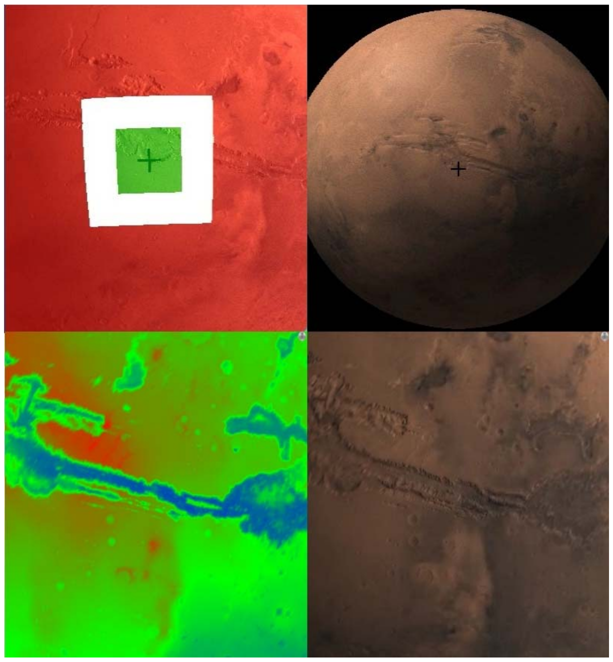
Figure 11. Dspace with DspaceTerrain clipmap shader renderings for the entire Mars MOLA data set. Clockwise from top left: false color showing clipmap levels, textured full Mars, Valles Marineris false color height map and textured.

Dspace and DspaceTerrain development is on going and in the future will include features to support the management and increased use of GLSL shaders for terrain, vehicle and environment rendering. Other possible uses for shaders for generating engineering data will be explored.

Dspace will also be extended to provide enhanced user interface support for interacting with Dshell++ simulations and for displaying simulation generated engineering data. Possible new use cases include roving vehicle traverse planning, designing (or editing) vehicle configurations, directing the placement of vehicle arm end effectors by selecting elements in the rendered 3D scene and communicating scene changes back to the simulation.

Acknowledgement

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

The authors would like to express their thanks to the members of the DARTS Lab team: J. (Bob) Balaram, Jonathan Cameron, Matt DiCicco, Rudranarayan Mukherjee, Hari Nayar and Leonard Reder, for their valuable input and many thoughtful discussions on this topic.

REFERENCES

[1] C. Lim, A. Jain. “Dshell++: A Component Based, Reusable Space System Simulation Framework” SMC-IT 2009, Pasadena, CA July 2009

[2] “Blender”, URL: http://www.blender.org/

[3] “OGRE (Object-Oriented Graphics Rendering Engine) “, URL: http://www.ogre3d.org

[4] “Coin3D”, URL: http://www.coin3d.org

[5] A. Jain, J. Balaram, J. Cameron, J. Guineau, C. Lim, M. Pomerantz, G. Sohl, "Recent Developments in the ROAMS Planetary Rover Simulation Environment" in IEEE 2004 Aerospace Conference, Big Sky, Montana, March 2004.

[6] J. Balaram, R. Austin, P. Banerjee, T. Bentley, D. Henriquez, B. Martin, E. McMahon, G. Sohl, "DSENDS - A High-Fidelity Dynamics and Spacecraft Simulator for Entry, Descent and Surface Landing," in IEEE 2002 Aerospace Conference, Big Sky, Montana, March 2002.

[7] “The Python Programming Language”. URL: http://www.python.org

[8] “SGI OpenInventor”, URL: http://oss.sgi.com/projects/inventor/

[9] “The OpenGL Graphics Language”, URL: http://www.opengl.org

[10] “The Virtual Reality Modeling Language”, URL: http://www.web3d.org/

[11] “Z-Buffer Shadow Algorithm”, Skinner, M., URL: http://www.gamedev.net/reference/articles/article1300.asp

[12] “OpenGL Shading Language”, http://www.opengl.org/documentation/glsl/

[13] Perlin, K. 2002. "Improving Noise." Computer Graphics 35(3).

[14] “A Simplified Wrapper and Interface Generator (SWIG)”, URL: http://www.swig.org

[15] H. Nayar, J. Balaram, J. Cameron, A. Jain, C. Lim, R. Mukherjee, S. Peters, M. Pomerantz, L. Reder, P. Shakkottai, S. Wall, “A Lunar Surface Operations Simulator,” in Proc. International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2008), Venice, Italy, Nov. 2008.

[16] R Madison, M Pomerantz, and A Jain, "Camera Response Modeling and Verification in ROAMS," i-SAIRAS 2005, Munich, Germany, September 2005.

[17] A Jain, J. Cameron, C. Lim, J. Guineau, "SimScape Terrain Modeling Toolkit," Second International Conference on Space Mission Challenges for Information Technology (SMC-IT 2006), Pasadena, CA, July 2006.

[18] A. Asirvatham, H. Hoppe, “Terrain Rendering Using GPU-based Geometry Clipmaps”. GPU Gems 2, M. Pharr and R. Fernando, eds., Addison-Wesley, March 2005.

[19] “Mars Orbiter Laser Altimeter”, URL: http://mola.gsfc.nasa.gov/

[20] “High Resolution Imaging Science Experiment”, URL: http://hirise.lpl.arizona.edu/TRA_000873_1780