(Preprint) AAS 18-093

FLEXIBLE BASILISK ASTRODYNAMICS VISUALIZATION SOFTWARE USING THE UNITY RENDERING ENGINE

Jennifer Wood[*], Mar Cols Margenet[†], Patrick Kenneally[‡], Hanspeter Schaub[§] and Scott Piggott[¶]

Visualizing complex numerical simulations is a critical component of modern astrodynamics software tools. The spacecraft simulation may contain a large number of spacecraft states and simulation parameters that are more readily comprehended when presented in context in a three-dimensional visualization. Spacecraft location, orientation, and actuator states can be displayed relative to the location of celestial objects along with spacecraft configuration parameters such as size, sensor locations and orientations, or dynamic states such as flexing or slosh. Basilisk is an open-source astrodynamics simulation frame being developed by the University of Colorado Autonomous Vehicle Systems (AVS) lab and the Laboratory for Atmospheric and Space Physics (LASP). This paper presents a companion software solution which will receive a stream of Basilisk state messages and dynamically visualize these states using the Unity rendering engine. This graphics engine allows for the production of high quality visualizations without requiring the engineer to learn low-level graphics programming. The Unity integrated development environment facilitates the process of expanding or enhancing the visualization, including the creation custom simulation interface windows and view ports.

INTRODUCTION

Basilisk provides a powerful astrodynamics framework that is used to simulate the behavior of space vehicles under flight-like conditions. The software package is architected as a series of C/C++ models wrapped around a message-passing interface (MPI) that interact with each other by exchanging structured data through this MPI.[1] These messages can be queried, logged, and analyzed after a given simulation run by plotting data out into the Python-level user interface. While data plots can be a convenient method for analysis, it is also important to be able to display the dynamic states of the spacecraft during runtime so that the system behavior can be visualized by the user, especially for real-time applications.

There are many examples of design and simulation systems that also support 3D renderings of the dynamic behavior of the astrodynamics system under test. One of the most widely-used packages is the Systems Tool Kit (STK) from AGI*. STK is used frequently in university and in industry

[*]Graduate Student, Computer Science Department, University of Colorado Boulder.
[†]Graduate Student, Aerospace Engineering Sciences, University of Colorado Boulder.
[‡]Graduate Student, Aerospace Engineering Sciences, University of Colorado Boulder.
[§]Professor, Glenn L. Murphy Chair, Department of Aerospace Engineering Sciences, University of Colorado, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO 80309-0431. AAS Fellow.
[¶]ADCS Integrated Simulation Software Lead, Laboratory for Atmospheric and Space Physics, University of Colorado Boulder.
*http://www.agi.com/products/engineering-tools

contexts to model spacecraft trajectories and render mission-level behavior. STK is a commercial product and requires users to purchase a license to use the visualization, which is largely tied to the underlying dynamics modeling of the STK application. It is privately-developed and uses a closed-source development model that requires all changes to the source code be approved and integrated by AGI, complicating customization by the user.

A public tool similar to STK is the General Mission Analysis Tool (GMAT) designed at the Goddard Space Flight Center[*]. GMAT offers many of the same basic capabilities as STK, with lower fidelity in some areas like attitude kinematics and optimization. It is released as a cross-platform (Windows, OSX, Linux) open-source application that the user can tailor as desired. The underlying graphics package uses WxWidgets which is a cross-platform GUI Library that offers bindings in a number of programming languages, but is not primarily focused on 3D rendering and visualization[†]. Like STK, it is difficult to extricate the spacecraft visualization system from the underlying vehicle simulation software.

Engineering DOUG (Dynamic Onboard Ubiquitous Graphics) Graphics for Exploration (EDGE)[‡] is designed to take dynamic state updates from an existing simulation. Often this is a vehicle simulation running inside NASA's Trick software package[§] with an associated NASA-developed dynamics modeling package like the JSC Engineering Orbital Dynamics package[¶]. Unlike STK or GMAT, EDGE is primarily designed to run locally on a Linux desktop with sufficient local resources to run the desired simulation and its visualization. While the EDGE framework is similar in intention to Basilisk, the licensing model can be problematic for general users as it can only be released for projects supporting the United States government.

Cosmographia is a rendering engine that is designed to display the Solar System and any desired spacecraft operating therein. A SPICE-enhanced version is available as a free download that the user can augment with their own spacecraft SPICE kernels to display the translational and rotational dynamics of their vehicle [‖]. There are also downloads provided for most modern operating systems (Windows, OSX, Linux) in a convenient binary format. Cosmographia is closed-source, limiting user extension and adaptation beyond the main interface mechanism of SPICE kernel files. This limits Cosmographia's use in distributed networks and requires the user to continuously output SPICE kernels for their spacecraft.

All of these systems provide accurate 3D renderings of spacecraft behavior with some limitations. The first hurdle can be the software licensing with expensive up-front cost (STK) or limited to approved use-cases (EDGE). The second is that customization of the display objects is either impossible (STK and Cosmographia) or difficult (GMAT and EDGE). Finally, in all the existing frameworks discussed it can be challenging to send dynamics and state information from a non-integrated application to the visualization system.

The original Basilisk Visualization utilizes the OpenGL Application Programming Interface (API)[**]. OpenGL is supported on any platform that implements that API and is concerned only with render-

[*] http://gmatcentral.org/

[†] http://www.wxwidgets.org

[‡] http://software.nasa.gov/software/MSC-24663-1

[§] htt://www.nasa.gov/centers/johnson/techtransfer/technology/MSC-24492-1-sim-toolkit.html#apps

[¶] http://www.univelt.com/FAQ.html#SUBMISSION

[‖] http://naif.jpl.nasa.gov/naif/cosmographia.html

[**] http://www.opengl.org/

ing 2D and 3D graphics. It does not directly provide any high-level implementations like animation or Graphical User Interface (GUI). The flexibility of OpenGL allows for the creation of a visualization that can graphically display astrodynamics simulation state messages for bodies, instruments and actuators. The major drawbacks of implementing the visualization in OpenGL are 1) that it is a relatively low-level language requiring intensive development of each visualization component to build the application and 2) the developer is responsible for managing the application's context to support multiple platforms.

To gain high-performance graphics with reduced barriers to user-driven extension and adaptation, development of the Basilisk Visualization shifted to the investigation of cross-platform game development engines like Unity* and Unreal Engine†. These game development platforms have been made freely available for public use provided the end-use revenue remains below a specified threshold. Unity and Unreal provide high performance drag-and-drop components like cameras, Graphical User Interface (GUI) objects, lighting, and shaders that can be reused and adapted to generate high-end graphics rendering. Spacecraft and component 3D models can be imported in a variety of formats, allowing for easy customization. Both Unity and Unreal support multiple platforms including Mac, Linux, Windows, iOS and Android and allow the user's application to be published to the desired platform with the click of a button.

This paper investigates the use of a high performance game development platform, Unity, to improve the capabilities of the visualization and simplify both cross-platform support and user-driven rendering of vehicle simulations. The Unity architecture provides adaptable off-the-shelf components that are designed to operate at the high performance levels required by state-of-the-art video games. The Unity game engine maintains an excellent tutorial collection and supports a robust online community that helps resolve the difficulties a new user may encounter and is an important consideration in choosing this platform for the Basilisk Visualization.

This paper introduces the Unity Basilisk Visualization framework and discusses approaches applied to address constraints created by using a preexisting graphics rendering framework. First, this paper describes how messages passed from Basilisk are imported to the visualization using low level custom C++ plug-ins. Second, the paper discusses how those messages are interpreted by the Basilisk Visualization and used to instantiate objects and display their current state. Next, the paper discusses the problems inherent in rendering objects over the large distances of space with the accuracy to support engineering and scientific applications, including spacecraft formation flying, and the solution approach Basilisk Visualization undertakes. Finally, this paper details how Unity prebuilt components and classes can be adapted to render complex scenes and provide cross-platform GUI support and intuitive use.

COMMUNICATION WITH BASILISK

Basilisk utilizes a Message Passing Interface (MPI) to internally relay messages between modules.[2,3] This provides flexibility within the framework that is then extended beyond the simulation to an external application, the Basilisk Visualization. The simulation can either archive the messages in a binary data file (allowing for easy sharing of scenario results) or pass them in real-time via the Black Lion messaging module.[4,5] Other platforms including FlatSat test beds,[6] in-test or even in-flight spacecraft can quickly implement adapters to their telemetry streams and relay the

*http://unity3d.com

†http://www.unrealengine.com/

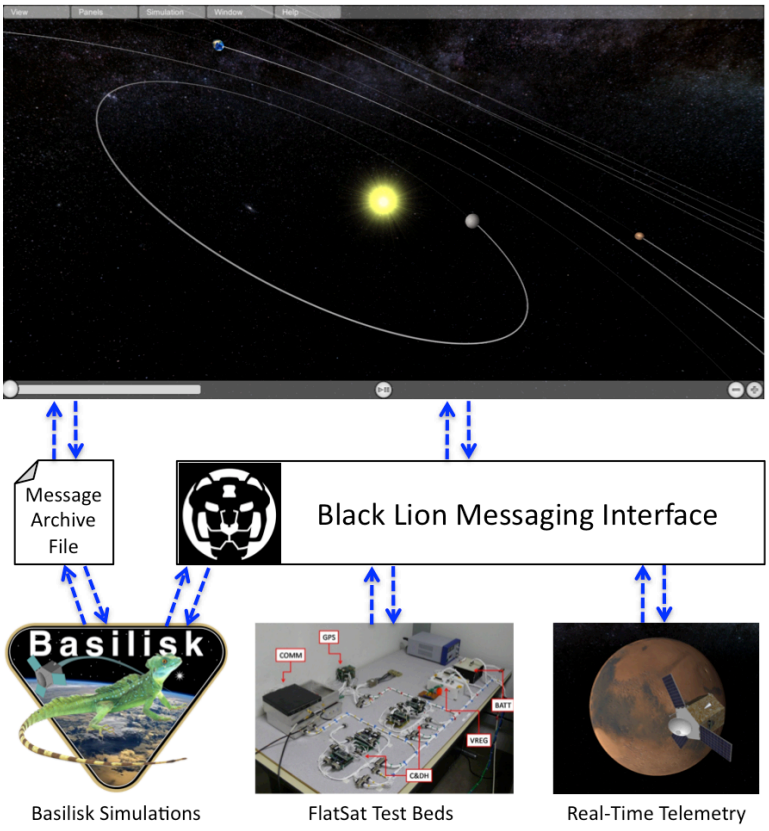

Figure 1. Possible Applications of Basilisk Visualization

spacecraft data to the visualization framework in the expected format as illustrated in Figure 1.

The Unity framework does not directly support Basilisk's native Python language. Unity does allow the use of low level native Plug-ins and the Basilisk Visualization exploits this capability by using custom C++ plug-ins for message handling. The custom plug-ins use the Simplified Wrapper and Interface Generator (SWIG) to convey the simulation data to the Unity C# scripts.

VISUALIZATION INITIALIZATION

The visualization uses the messages passed from Basilisk to determine what bodies, sensors, and actuators should be initialized in Unity for the current scenario. Only components whose messages are available will be rendered. For example, the simulation of a simple Earth-Moon system will likely only include those two bodies and their state messages and will result in the creation of only those bodies in the visualization. Rendering only the components with available state messages provides the user with quick visual confirmation of their desired scenario set-up. Likewise, only spacecraft components modeled in the current Basilisk run will have GUI subpanels instantiated and enabled.

Unity encourages the use of prefabricated components that can be instantiated or redeployed as required while the application is running. The visualization uses the Unity prefabricated game object (Prefab) design concept and provides generalized game object templates that are customized as needed. These Prefabs can be used as templates for customization by Basilisk users to adapt the

visualization to support their needs.

At present, these Prefabs include a generic celestial body, spacecraft, orbit line, onboard camera, and GUI subpanel. Additional prefabricated templates will be added as needed to ease extension by Basilisk users. The Basilisk Visualization *GameController* object reads the Basilisk messages and orders specialist submanagers to create and control game component instances as required. Each submanager object is responsible for a discrete portion of the visualization framework as as shown in Figure 2.

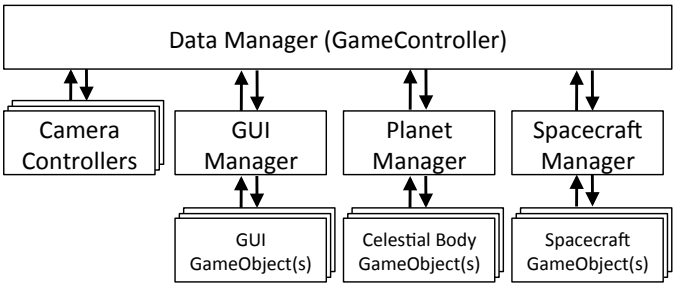

Figure 2. Basilisk Visualization Component Communication Diagram

The subcomponents are flexible and can adapt to various simulation scenarios from basic 2-body problems, to simple models of the solar system, to full simulation runs with multiple celestial bodies, spacecraft, and their attendant sensors and actuators.

A PROBLEM OF SCALE

Industry standard Game Engines like Unity and Unreal use 32-bit floating point precision to drive the graphics rendering. This leads to difficulties like camera jitter when distances are large and the remaining available precision is small. The limitations of floating point precision must be addressed when rendering interplanetary distances and disparate scales and is tackled by the Basilisk Visualization with two techniques: multiple views and camera layering.

Multiple Views

To render a view of a system as large as the solar system, or as small as spacecraft flying in formation, the Basilisk Visualization employs multiple views. The visualization starts in solar system view with distances scaled by 1:1E8 meters. Planets are drawn at an artificial scale for easy navigation by the user to areas of interest [Figure 3]. The user can double-click on a planet to change the camera target and automatically zoom in to a local planet view.

In local planet view, the planet and any local bodies (moon and/or spacecraft) are visible. The local celestial bodies and the orbits of any secondary bodies are drawn to the scale of 1:1E5 meters, but spacecraft in the view are shown at an artificially large scale for visibility. Selecting a spacecraft as the camera target results in a transition to the local spacecraft view.

In local spacecraft view, the spacecraft must be drawn 1:1 meter to allow for placement of sensors and actuators on the spacecraft body and for accurate representation of spacecraft flying in formation. At this scale, rendering the spacecraft in the same frame with its parent body becomes impossible due to floating point precision constraints and camera layering must be employed to correctly render the scene.

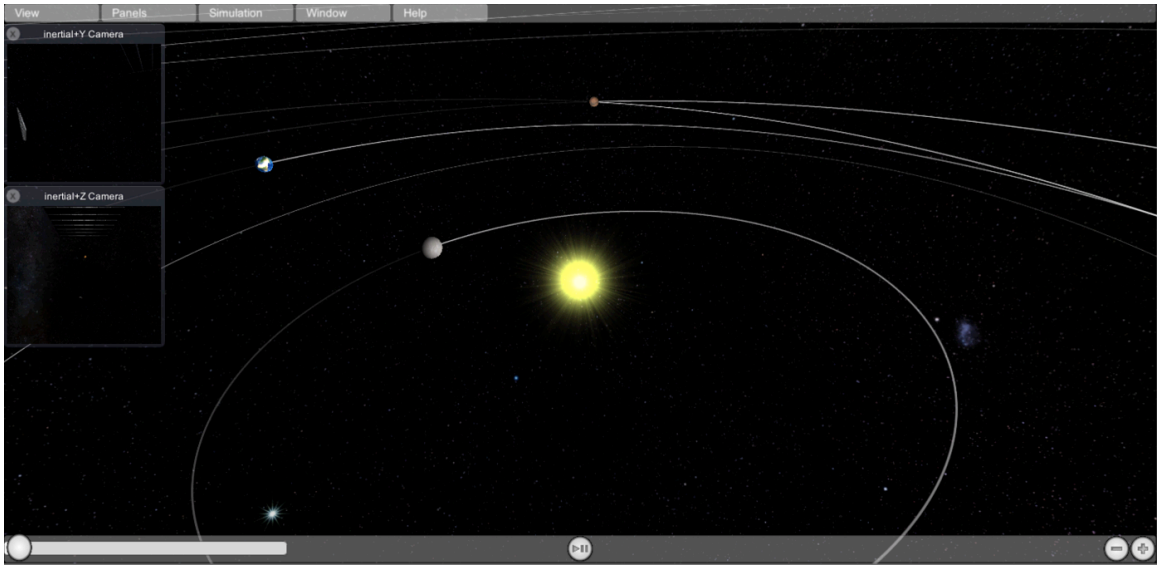

Figure 3. Basilisk Visualization in Solar System Scale View

Camera Layering

To solve the problem of rendering detailed models and positions with an accurate representation of relatively nearby celestial bodies, the Basilisk Visualization takes a page from modern space games and layers the view from a local camera of the true-to-scale spacecraft over a dynamic skybox in Figure 4. The main camera provides the dynamic skybox texture by continuing to render the local planet view. The main camera is fixed to the center of the spacecraft position in that frame, but its rotation is commanded by the spacecraft local camera so that both cameras always point in the same direction.

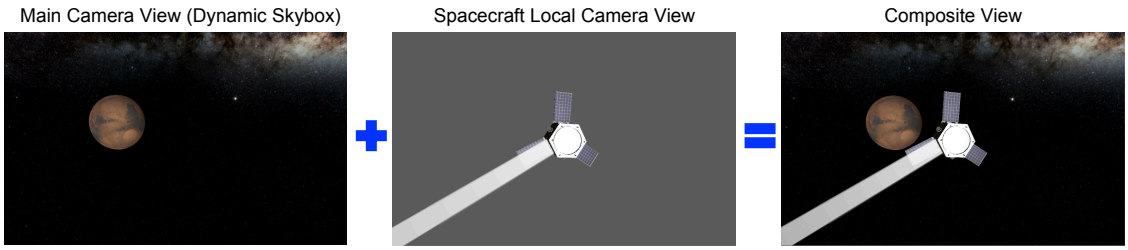

Figure 4. Simultaneously Rendering Detailed Local and Distant Views with Camera Layering

By layering the two views, the user can visualize local spacecraft behaviors and compare them with the spacecraft position relative to the local body. This complex view rendering is made easy to implement by the built-in Unity camera component and its supported methods and settings. The result avoids floating point problems like camera jitter and provides a beautifully rendered background view, allowing the user to focus on the detail the local spacecraft view provides.

SPACECRAFT BEHAVIOR VISUALIZED

The goal of the Basilisk Visualization is to convey as much information as possible about the underlying numerical simulation in a graphical way. This data is communicated in a number of ways, including in the rendered visual models for the planets, moons, and spacecraft present in the view. Graphical overlays to draw local coordinate frames and orbit lines indicating a body's position and direction of motion are available for user reference and can be turned on or off as desired. Information from the spacecraft sensors, cameras, and actuators can be displayed in GUI subpanels that can be moved, resized, and toggled on and off. The visualization allows for playback control to better examine spacecraft behavior or a critical stage in a mission like the Mars Orbit Insertion (MOI) illustrated in Figure 5.

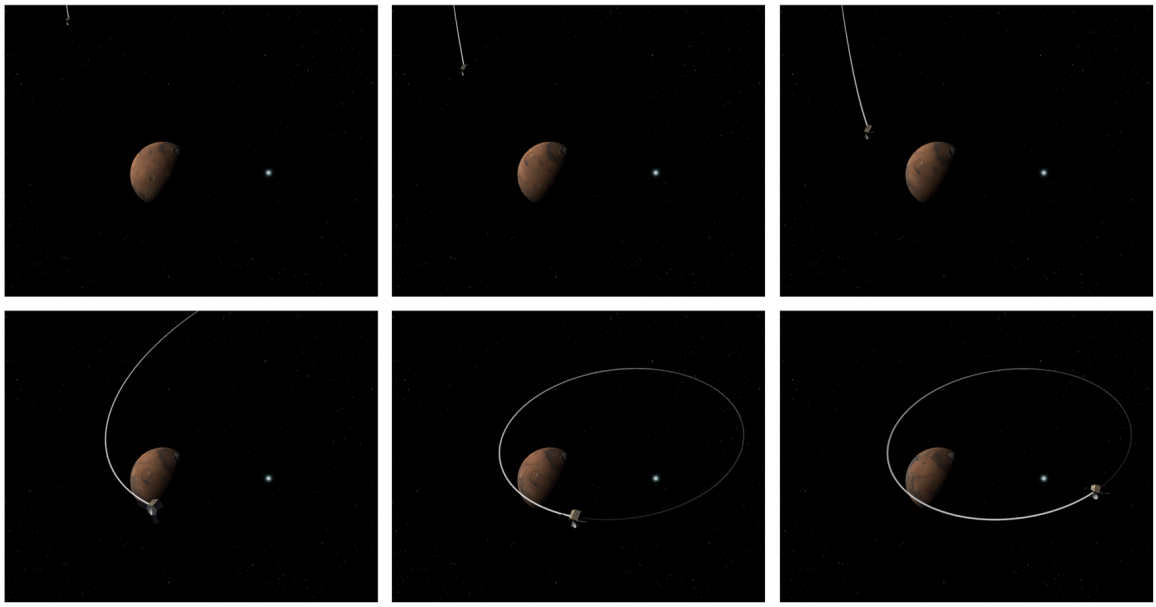

Figure 5. Basilisk Visualization of Basilisk Simulated Mars Mission Orbit Insertion

Coordinate Frame Overlays

The Basilisk Visualization can display local coordinate frame axes for celestial bodies (planets, moons, etc.) and spacecraft. The base class for the coordinate axes display calls the openGL graphics layer directly to generate pixel-width lines that are drawn after all the higher-level Unity rendering completes. The coordinate axes base class can be reused to provide sensor and actuator coordinate frame references in the local spacecraft view.

Orbit Lines

The orbital elements of each body in the simulation are calculated and updated at every time step. An orbit line for each body needed in the current scene is rendered using the LineRenderer Unity component. Each orbit line graphically indicates the current position of its body and its direction of travel by reducing the opacity of the line along its length with the boldest portion of the line directly behind the bodys current position.

Subpanel Views

The Generic Subpanel Prefab can be reused to provide heads up display of spacecraft information including visual representations of the current states of actuators and sensors in the simulation. These subpanels can also display additional camera views along desired orientations that can be preset (Spacecraft +Y, Spacecraft +Z, etc.) or aimed as desired by the user. The field of view of these additional cameras can be adjusted and/or a bore sight mask can be applied to the view to reflect instrument specifications. These camera views are rendered to Generic GUI subpanels that can be moved, resized, and toggled on and off.

The messages for actuators, like thrusters and reaction wheels, as well as sensors, including sun sensors, star trackers, and gyroscopes, can also be supported with GUI subpanels conveying information to the user. Only actuators and sensors whose messages are provided by Basilisk are instantiated in the current visualization which can provide quick confirmation to the user of the validity of their scenario settings.

Playback Control

On opening the binary message file or a datalink with Black Lion, the visualization tasks a thread with maintaining a continuous state archive for the entire current simulation. This archive is employed so that the user can interact with the view in front of them without changing the runtime behavior of the simulation under test. The visualization provides GUI controls to adjust the current time step so the user can easily revisit an event of interest during the run. The user can increase or decrease the visualization rate from the default of one time step/rendered frame with the playback controls.

The granularity of the rendered video reflects the sample rate of the Basilisk messages and can adjusted by increasing or decreasing the size of the time step between Basilisk solutions. If the user has imported an archive file from a previously run simulation, they can examine all of the observed behavior of that test and loop through the messages repeatedly. While the visualization is running, the user can adjust the cameras, switch view scales, and enable or disable subpanels. This flexibility is available to the user whether the messages come from an archived file or from an an in-progress simulation because the archive thread continues to log the state data generated by Basilisk.

PLATFORM INDEPENDENCE

Basilisk is a cross-platform supporting framework and the Basilisk Visualization must also be available to those users. Unity provides support for multiple platforms including PC, Mac, and Linux. The Unity Editor allows the user to publish the Basilisk Visualization project to an application targeting these platforms and also allows the publisher to determine the desired rendered video quality. The resulting application is a binary file with all dependent assets included.

The Basilisk Visualization implements the Unity Event System and User Interface methods and classes. This allows the visualization to take advantage of Unitys robust event system to send messages within the application to trigger behaviors commanded by the user through a touch interface or mouse control regardless of the user's platform. The Unity UI system allows for drag and drop wiring of buttons, sliders, and other UI components to their desired method calls and can be easily extended to support additional functionality desired by a Basilisk user.

FUTURE WORK

The inclusion of a detailed star catalog to generate the background skybox will allow users to check the modeling of instruments such as star trackers and onboard cameras against observed results. Adding a planetary ephemeris will allow for the generation and motion of celestial bodies not modeled in the Basilisk simulation if desired. The visualization generated data will be clearly delineated from those entities generated from Basilisk messages to continue to allow for visual error checking of the scenario settings.

REFERENCES

[1] J. Alcorn, H. Schaub, S. Piggott, and D. Kubitschek, "Simulating Attitude Actuation Options Using the Basilisk Astrodynamics Software Architecture," *67th International Astronautical Congress*, Guadalajara, Mexico, Sept. 26–30 2016.

[2] M. Cols Margenet, H. Schaub, and S. Piggott, "Modular Attitude Guidance Development using the Basilisk Software Framework," *AIAA/AAS Astrodynamics Specialist Conference*, Long Beach, CA, Sept. 12–15 2016.

[3] M. Cols Margenet, H. Schaub, and S. Piggott, "Modular Platform for Hardware-in-the-Loop Testing of Autonomous Flight Algorithms," *International Symposium on Space Flight Dynamics*, Matsuyama-Ehime, Japan, June 3–9 2017.

[4] M. Cols Margenet, P. Kenneally, H. Schaub, and S. Piggott, "Black Lion: Software Spacecraft Simulation Architecture Joining Heterogeneous Components," *10th Workshop on Spacecraft Flight Software*, The John Hopkins University Applied Physics Laboratory, MD, December 4–8 2017.

[5] M. Cols Margenet, P. Kenneally, and H. Schaub, "Software Simulator for Heterogeneous Spacecraft and Mission Components," *AAS Guidance and Control Conference*, Breckenridge, CO, February 2–7 2018.

[6] S. M. Toro, J. Greenbaum, T. Campbell, G. Holt, and G. Lightsey, "The FASTRAC Experience: A Student Run Nanosatellite Program," *24th AIAA SmallSat Conference*, Logan, UT, August 2010.