Applied Multi-Mission Telemetry Processing and Display for Operations, Integration, Training, Playback and Event Reconstruction

Marc Pomerantz[1], Viet Nguyen[2], Daren Lee[3],Christopher Lim[4], Tom Huynh[5]
Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109

Conveying spacecraft health and status information to mission engineering personnel during various mission phases, including mission operations, is a requirement to achieve a successful mission. For NASA/JPL spacecraft, that often means displaying hundreds of telemetry channels from a variety of sensors and components emitting data at rates varying from 1hz-100hz (and faster) in a way that allows the operations team to quickly evaluate the health of the vehicle, identify any off-nominal states and resolve any issues. In this paper we will discuss the system design, requirements and use cases of three telemetry processing and visualization systems recently developed and deployed by our team for NASA's Low Density Supersonic Decelerator (LDSD) test vehicle, NASA's Soil Moisture Active/Passive (SMAP) orbiter, and JPL's Sampling Lab Universal Robotic Manipulator (SLURM) test bed.

Nomenclature

MSL	=	NASA's Mars Science Laboratory rover
EDL	=	Entry, Descent and Landing mission phase
SMAP	=	NASA's Soil Moisture Passive Active spacecraft
LDSD	=	NASA's Low Density Supersonic Decelerator test vehicle
SLURM	=	JPL's Sampling Lab Universal Robotic Manipulator
DARPA	=	Defense Advanced Research Projects Agency
Qt	=	Toolkit for building UI applications
PyQt	=	Python wappers for Qt
PyQwt	=	Python wrappers for plotting widgets that extend Qt
CSV	=	Comma Separated Values
UI	=	User Interface
IMU	=	Inertia Measurement Unit

I. Introduction

Conveying spacecraft health and status information to mission engineering personnel during various mission phases, including mission operations, is a requirement to achieve a successful mission. For NASA/JPL spacecraft, that often means displaying hundreds of telemetry channels within context, from a variety of sensors and components emitting data at rates varying from 1hz-100hz (and faster) in a way that allows the operations team to quickly evaluate the health of the vehicle, identify possible off-nominal states and resolve any issues detected.

In this paper we will discuss the system design, requirements and use cases of three telemetry processing and visualization systems recently developed and deployed by our team for the first two flights of the Low Density Supersonic Decelerator (LDSD) test program, the Soil Moisture Active/Passive (SMAP) orbiter, and the Sampling Lab Universal Robotic Manipulator (SLURM) test bed. All deployed systems were developed using Object-

[1] Member of Technical Staff, Robotics Modeling and Simulation Group, Mail Stop 198-235, AIAA Member.
[2] Member of Technical Staff, Robotics Modeling and Simulation Group, Mail Stop 198-235, AIAA Member.
[3] Research Technologist, Robotics Modeling and Simulation Group, Mail Stop 198-235, AIAA Member.
[4] Member of Technical Staff, Robotics Modeling and Simulation Group, Mail Stop 198-235, AIAA Member.
[5] Member of Technical Staff, Simulation and Support Equipment Group, Mail Stop 198-138, AIAA Member.

Oriented design principles, the Python Language[1] and the Qt[2] UI framework. In addition, all applications share a common code-base with a mission specific-layer and were multi-threaded to support multiple tasks, such as the concurrent processing of multiple telemetry data streams, data logging, and distributing data to single or multiple visualization platforms. Servers in our deployed systems where either custom Python software based developed by our team or open source message passing services.

II. Background

Leveraging on the telemetry visualization software that our team developed for the live 2012 Mars Science Laboratory (MSL) entry, decent and landing (EDL) night event[3], the simulation-based visualization of the LDSD test vehicle's predicted trajectory and flight performance, and telemetry visualization and robot control visualization developed for the JPL Robodome robots as shown in Figure 1, our team has developed a cost effective architecture and re-usable software framework to support the processing, serving and visualization of spacecraft telemetry for a variety of data rates, including much higher rates (up to 1000hz) than the 4hz rate we encountered during the MSL EDL, while presenting important mission information to operations personnel in an easy to understand, visually compelling, and organized format. Recent past work also supported the 2010 MoonRise proposal, telemetry visualization and control displays for JPL Maritime R & D tasks as well as visualization for JPL and DARPA robotic vehicle simulation efforts.

Figure 1. Past Supported Visualization Efforts

III. System Design and Methodology

Starting with a tool set based on Python, PyQt[4] and PyQwt[5] we designed and built an underlying processing framework and display solution with re-useable Python classes used to instantiate UI elements for scalar, plot and mapping widgets, display windows, asyncronous data processing and display using multiple threads of execution, and socket-based I/O utilizing TCP/IP and UDP network protocols. All of the described telemetry visualization systems were based on a client-server model where telemetry was either provided directly from the vehicle to our local server, with decommutation provided by a dedicated processing unit or via the JPL Ground Data System.

Taking a cue from past work we performed at JPL, and in industry designing display list-based, 3D rendering engines for space mission and robotic vehicle simulations and mission operations, we developed a novel method for associating telemetry channel attributes and actual telemetry data received, along with the UI widgets used to display the telemetry data, by building a display list containing entries representing each telemetry item to be displayed. As depicted in Figure 2, we use documents provided by our flight project customers, typically in CSV format, we extract individual channel attributes such as channel name, identifier, high and low limits and any scaling factors or equations, and then auto-generate Python dictionaries containing this channel attribute information, as well as references to functions used to scale the channel data and also reference to the actual Qt widget that will display the continuously changing channel data. The associated Qt widgets are either specified explicitly

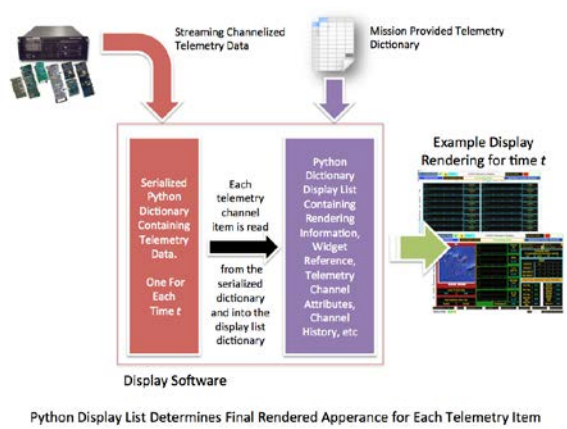

Figure 2. System Flow and Display List Rendering

when we use Qt Designer or for large numbers of similar widgets, we have written Python scripts to auto-build UI elements.

At run time and during the display rendering pass, the display code main thread iterates over the entire set of dictionary entries (representing the complete set of telemetry channel w/attributes to be displayed) and pulls the indicated channel value from the most recent serialized telemetry data received from the spacecraft or robotic system. For each telemetry channel item on our display, we render either scalar or trend values (plots) with color indicators to highlight within limits or in alarm. On a separate thread, telemetry channel data is continuously read from our server either to be immediately displayed when in "live" mode or saved in the display list for later recall by the user when paging backward and forward over time in "playback". While we acknowledge that the memory usage of an application will continuously grow during system execution, we've found that having access to all the telemetry data received during a run is important to our customers, especially when the need to characterize system performance over time (trends) is desired for data such as bus voltages, thermal sensor values and IMU samples. Additionally, because we're storing all received telemetry in the display memory, we can log the entire telemetry data to file for replay at a later time.

IV. LDSD Flight Operations Displays

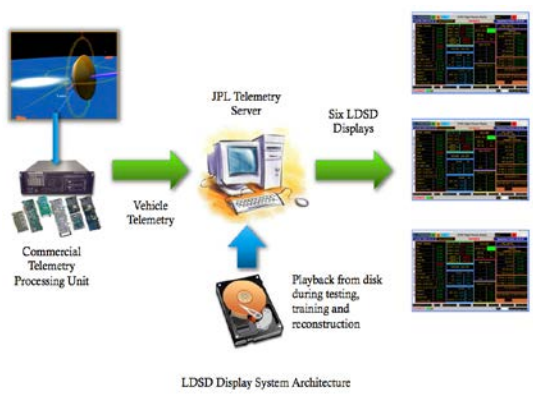

Figure 3. LDSD Telemetry Display System Architecture

In support of the LDSD mission, we designed a client-server system (Figure 3) to process the 200-300hz rate test vehicle telemetry data and provide a single-point telemetry processing capability that could serve multiple operations displays, all located on a closed and secure network. Because the LDSD flight tests were conducted at the Pacific Missile Range Facility (PMRF) we did not have access to the JPL institutional Ground Data System (GDS) typically used to serve channelized telemetry data. So, with a single workstation-class server computer system doing the heavy lifting, we processed the packetized telemetry stream from the test vehicle (and decommuted by a commercial processing unit), converted that stream to channelized telemetry and then served that telemetry in a serialized Python dictionary format, via a network swtch, to the multiple (six), and relatively light-weight, and inexpensive display clients running on Apple Mac Mini computer systems. While we've described the clients as light-weight, they were required to perform operations to render data to the screen, log data to file, execute splashdown prediction models, graph certain data, maintain telemetry data in memory and then recall that data when the user scrolls the display back and forth over time all while continually reading telemetry data over ~~a~~ TCP/IP and UDP sockets. To allow for these simultaneous operations each display client is a multi-threaded application with execution threads that perform the operations described above. We found that the Mac Mini systems, with Intel i5 CPUs and 8 GB of RAM, were capable of maintaining display data rates of ~20hz.

Figure 5. LDSD Flight Dynamics Pages

For the LDSD Flight Director page (Figure 4) we worked closely with the 2014 and 2015 Flight Directors to

best present an overall picture of the test vehicle's health based on data from a variety of subsystems that included Power Bus voltages and currents, IMU, GPS, Balloon connector, cameras and digital video recorder (DVR) states. As the test vehicle's systems were powered up, the Flight Director page would display green values for "not in alarm" telemetry items. During the pre-launch checkout, the Flight Director looked for a completely green board, which confirmed that the test vehicle was in a proper state for flight. Our LDSD Displays also contained multiple pages designed to present collections of data that support the Flight Director, Thermal Engineer, Flight Dynamics Engineer and Flight Reconstruction Engineer. The Flight Dynamics pages (Figure 5) contained vehicle performance information along with a situational awareness map display of the launch and test flight region in the Pacific. Overlayed onto the map and continuously updated during flight, were rendered icons showing test vehicle location and attitude, and predicted splashdown ellipses based on vehicle altitude. Integrated into the these pages was the ability for the Flight Dynamics engineer to initiate the execution of a splashdown prediction model (provided by the LDSD project) to be run typically after parachute deployment, that used vehicle performance data as received int the telemetry stream. Upon running the splashdown prediction model, an icon representing the predicted splashdown location was rendered onto the map, Overall the LDSD telemetry displays that we developed processed and displayed over 200 individual items, including over 100 plot items from four simultaneous telemetry streams.

A. Integration and Test

During development of the LDSD displays, we quickly realized that the display software could be valuable to the spacecraft integration and test team as the display software was developed prior to full integration of the test vehicle hardware. During on-going vehicle testing, the LDSD operations team could quickly identify if components were were not powered up in sequence, or if, for example, a thermal sensor was not wired in to the system. And while we did not have the time or budget, we discussed and did preliminary design work for a future, telemetry-based 3D contextual display that would display vehicle component telemetry along with the actual correct placement of components onto a 3D model representing the LDSD test vehicle. This contextual display could help the operations team determine that sequences designed to fire thrusters in a certain order, would actually be firing the thrusters mounted at the required locations on the vehicle.

B. Training

Prior to both LDSD test flights, dress rehearsals were conducted with operations personnel at the test flight facility and with the complete ground data system. During these tests, playback and simulation data was piped through the ground system and into our LDSD displays, thereby giving operations personnel an opportunity to view displays and data very similar to what they would see on flight day. Again, due to time and budget constraints, we discussed, but were not able to build a fault injection system that would have allowed LDSD operations experts to design training scenarios with simulated faults injected into the telemetry data, up stream of the LDSD displays. When developed, this type of simulated fault system could be used to script training exercises with a combination of predictable or randomly injected simulated faults. For a multi-year flight test project like LDSD, this could be a cost savings and reliability tool when training up new operations personnel, year over year.

C. Playback and Reconstruction

The display software is capable of logging onto the local display computer which gives LDSD operations engineers the ability to playback the actual, logged data from test flights immediately. If desired, it was an easy process to copy the log data from the display computer to LDSD engineer's personal laptop thereby allowing the flight reconstruction analysis to begin almost immediately at the conclusion of the test flights.

V. SLURM Arm TlmViz

In support of JPL's Sampling Lab Universal Robotic Manipulator (SLURM) effort, we were tasked to develop a client-server system to access high-rate data from the JPL-developed SLURM arm robotic system targeted for deployment on the NASA Mars 2020 rover and graph various telemetry channels for the duration of a test. Project engineers requested that the TlmViz system be designed to plot up to ten channels during SLURM Arm test and development with tests durations possibly lasting up to a few hours. The high-rate data from the various sensors and actuators on the SLURM system range up to approximately 1000 Hz so it was a challenge to serve and display data

at those rates. To meet the specified requirements, we developed a client-server system using Python, PyQt4, PyQtGraph[6] and RabbitMQ[7]. The RabbitMQ server publishes telemetry data as requested by the user at system start up, and the TlmViz application subscribes to the appropriate Topics[2]. Because our RabbitMQ server publishes all available telemetry items, multiple instances of our TlmViz application can be run on different laptops or workstations, with each receiving identical or unique sets of telemetry items as specified by TlmViz configuration files loaded by the user at startup time. As we described in section III above, the TlmViz software is multi-threaded with the main thread handling operations related to setting up and rendering the multiple telemetry graphs and the second thread continuously reads streamed telemetry data from the RabbitMQ server. When topic items are received by the application, each is placed in the Python Queue owned by the associated plot. Using Python Queues allows us to asynchronously update the plots internal data structures without needing locks or semaphores. As we did on our LDSD software, this two thread architecture also allowed us to build in a feature that lets the user switch to "playback" mode and then page forward and back over time to view data received minutes or hours earlier. When in "playback" mode no data is lost as the telemetry thread continues to acquire data, with plots being updated when the user switches back to "live" mode.

While TlmViz was designed to process and display telemetry at rates at ~1000 Hz, the currently deployed version of TlmViz receives data at ~20 Hz rate based on decisions made by the operations team and determining that 1000 Hz data was not needed.

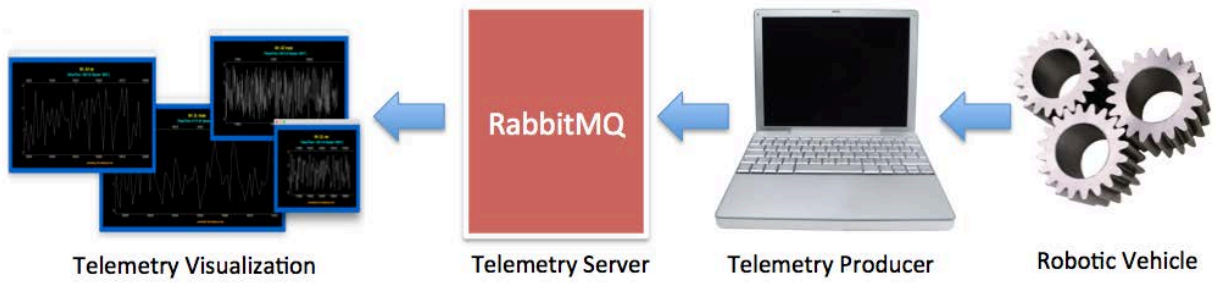

Figure 6. SLURM TlmViz Data Flow

Figure 6 shows the high-level data flow for our SLURM/TlmViz system. As described in a new paper submitted for publication[9], we found this publish/subscribe method to be fast and efficient and moving forward, we're looking at possibly replacing our older LDSD telemetry server with a Kafka[8] or similar messaging system.

VI. SMAP Critical Sequence Telemetry Visualization

NASA's Soil Moisture Active/Passive (SMAP) spacecraft engineering team sought out a telemetry visualization tool to support three of their critical mission sequences: reflector boom arm deployment, reflector deployment, and reflector spin up. The SMAP team needed a tool to gauge whether spacecraft behavior was nominal or off-nominal as well as present deployment or spin up subsystem specific telemetry in a way that mission specialists, management and the public could easily understand.

Leveraging on existing work with the LDSD telemetry display system (section III above), we were able to rapidly develop a SMAP-specific application framework and develop a single application for the SMAP operations team that was used for each of the three critical sequences as described below. Each version of the application displayed specific telemetry channels as requested by the SMAP operations team that indicated spacecraft health and state. In addition to the numeric and text fields displayed, a large visualization area was reserved for display of a visual representation of the spacecraft subsystem being monitored during the sequence. A soft indicator of progress was available in the telemetry stream and that was used to update a progress bar at the bottom of the displays. Estimates from the engineering team for events along this progress bar/timeline were included to help viewers better understand the sequence progress.

A. GDS To Display Communication

Unlike LDSD, SMAP used the JPL institutional Ground Data System (GDS). GDS contains a suite of command line tools designed to access stored and live telemetry. We were able to leverage our LDSD display work and re-use existing communication code to allow us to configure the display to receive telemetry data over UDP. By using a small custom Python script, we were able to ~~shuttle~~ route telemetry data from live or playback GDS sessions into multiple~~our~~ SMAP displays which allowed us to test and verify our software with the same telemetry plumbing that we would see during flight operations.

B. Boom Deployment

The first SMAP critical sequence supported was the deployment of the boom arm (Figure 7) which extended the reflector outward from the spacecraft body from a stowed configuration. The telemetry available during the sequence was insufficient to precisely determine the kinematic state of the arm, although the approximate state could be inferred from the available telemetry (e.g. cable tension, spool revolutions, etc.). Because we could not accurately and continuously predict the Boom kinematic state, we decided to use image frames from a pre-existing engineering simulation of the arm deployment, and keyframe those images to a sequence progress indicator. This greatly simplified V&V and provided a nearly equivalent visual result while reducing development time and cost.

Figure 7. SMAP Boom Deployment Telemetry Visaualization

C. Reflector Deployment

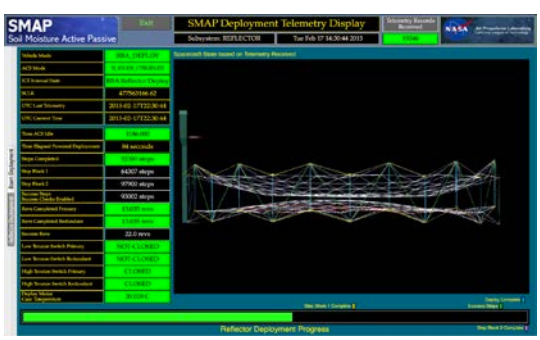

Figure 8. SMAP Reflector Deployment Telemetry Visualization

Reflector deployment was the second critical sequence we supported for SMAP. Visualizing the deployment of the reflector (Figure 8) was potentially more complicated than visualizing the boom arm because of the soft cloth-like dynamics. However, utilizing the same approach as the Boom Deployment, we were able to keyframe a series of images from an existing engineering simulation to telemetry items indicating deployment progress for the critical reflector deployment sequence.

Using images from existing simulation work and keyframing it to the progress telemetry proved to be an excellent design decision. More complex visualizations were of course possible but the operations team found the images to be understandable and familiar. The motions of the two

deployments on the spacecraft were essentially confined to a 2D plane so the lack of a 3D visualization was not detrimental and introducing more complexity or fidelity would have had diminishing returns at the cost of expanded V&V requirements.

D. Reflector Spin Up

With the reflector successfully deployed it needed to go from its static state to its nominal science spin rate of 14.6 RPM. This was done across two days and two critical sequences: an initial spin up to about 5 RPM on day one and a final spin up to the nominal spin rate on day two. We supported both of these sequences with a modified display developed under very tight time constraints.

For this sequence the engineering team was interested in not only the speed and state of the reflector but the kinematic state of the spacecraft as well, as the reflector motor spin up could cause a counter-active torque causing the spacecraft bus to tumble. Visualizing this properly was important to the SMAP operations team and we determined that the developed solution required a 3D visualization.

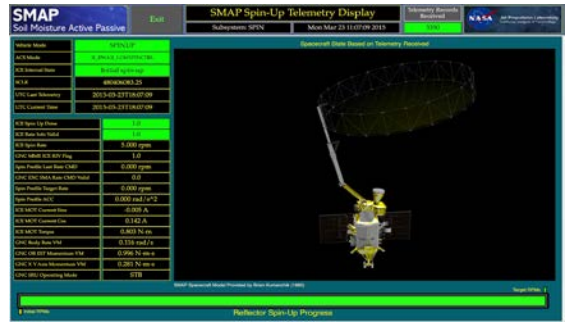

Figure 9. SMAP Reflector Spin Up Telemetry Visualization

We developed the 3D solution (figure 9) by leveraging existing Python code developed for the Boom and Reflector deployment displays and open source tools. An Education and Public Outreach (EPO) team at JPL had already built a 3D model of the SMAP spacecraft. This model was built natively in Blender[10], a free and open source 3D modeling and rendering software package. By taking advantage of Blender's rendering capabilities we knew that we could develop a Blender-based offsite renderer to support the SMAP spin up operations. This remote rendering design had numerous advantages. First, very little modification of the existing SMAP display visualization was needed because it already supported the display of images. Second, the 3D model being used had been constructed using Blender so no import, export, or model translation work needed to be done. Third, Blender's Python scriptability meant we could manipulate the model based on received telemetry and direct Blender to render to an image file. Fourth, Blender's expansive set of rendering functionality would be available for us.

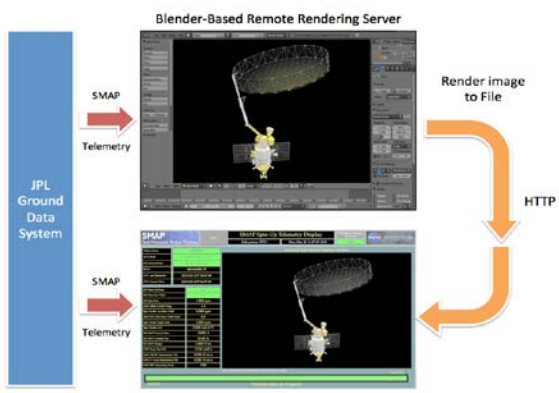

Figure 10. SMAP Reflector Spin Up Remote Rendering Data Flow

The data flow was modified so that the telemetry stream was split in two, with one stream going directly into the SMAP display to update the various scalar values, and one stream going to the remote Blender renderer. The Blender renderer ran on a separate workstation-class computer that received telemetry via UDP similar to the SMAP display. Images were rendered to image files and stored on the workstation local disk. Those image files were then served from the remote workstation to the SMAP display software using a simple Python HTTP server. The SMAP display retrieved these rendered images at a 4 Hz rate. Figure 10 illustrates this data flow design.

An interesting and beneficial side effect of this design was the ability to see the visualization stand-alone in a browser. With a simple refresh HTML page users were able to view the visualization on their mobile devices or computers.

VII. Conclusions

Multi-mission telemetry processing and display has been shown to be a valuable tool during vehicle integration and testing to operations and analysis and reconstruction, including the training of operations personnel for nominal and off-nominal cases prior to and after launch. We have demonstrated that a software framework consisting of Python/Qt can provide extensibily and performance for 2D and 3D display designs and is a cost effective solution, given the number of easy to use Python modules that are available for graphing data, managing multi-threaded and multi-process applications, OpenGL[11] integration, interprocess communication, and image and map display.

We plan to extend this work in the future to provide accurate contextual spacecraft visualization (from telemetry) during flight and spacecraft component visualization to be used by integration and test engineers to confirm that not only are the various spacecraft sub-systems are operating as designed, but that components are located correctly on the spacecraft chassis. An example would be attitude control system (ACS) thrusters and associated wiring and sensors installed in the correct location as well as correct operation of the flight software to control those ACS components.

Acknowledgments

The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

Proceedings

[3]Pomerantz, et al., "Multi-Mission Simulation and Visualization for Real-time Telemetry Display, Playback and EDL Event Reconstruction" AIAA SPACE 2012 Conference, Pasadena, CA

[9]Lee, Pomerantz, "Message Brokering Evaluation for Live Spacecraft Telemetry Monitoring, Recorded Playback, and Analysis", AIAA SPACE 2015, Pasadena, CA (submitted for publication)

Computer Software

[1]Python 2.7, http://www.python.org/

[2]QT 4.8. The Qt Company, http://www.qt.io/

[4]PyQt4, Riverbank Computing Limited, http://www.riverbankcomputing.co.uk/

[5]PyQwt, Gerard Vermeulen, http://pyqwt.sourceforge.net/

[6]PyQtGraph, Open Source, http://www.pyqtgraph.org/

[7]RabbitMQ 3.5, Pivotal, https://www.rabbitmq.com/

[8]Apache Kafka, http://kafka.apache.org/

[10]Blender 2.7, Blender Foundation, http://www.blender.org/

[11]OpenGL, OpenGL.org, https://www.opengl.org/