NASA Jet Propulsion Laboratory

Section 347D - Visualization

Ari Brown

Mentor: Marc Pomerantz

Part 1: 3D model COLLADA Format Loading Optimization

The Ranger platform is a 3D visualization software developed by JPL in order to accommodate the needs of space missions and earth projects by enabling data-driven visualization with precise accuracy. Ranger is a platform that allows users to build visualization applications with nothing more than outlining the basic components in a scene, and it makes visualization software accessible because it is built for browser. As an example, the Earth Web application built on Ranger shows real-time motion of spacecraft around earth. The assembly of these types of visualizations is made easy by Ranger's API.

When I arrived this summer, one issue Ranger was having was with the speed at which the platform could load COLLADA files. The COLLADA file format is a format which encodes the geometries and other parameters necessary to encapsulate 3D models. For models with many complex geometries, the current COLLADA importer would take upwards of a minute to load. For instance, the COLLADA file of the International Space Station (ISS), which has many complex sub-meshes, took about 57 seconds to load. Because of the slow loading time, the ISS was left out of applications like Earth Web.

The first step in approaching the problem of code optimization is profiling the old code. The profiling of the old COLLADA importer took about a week, between learning parts of the software architecture of the Ranger platform and testing where the bottleneck was.

The Ranger platform uses C++ to do fast math calculations needed for visualization. In order for Ranger to be a browser-based system, the C++ code is compiled by the Emscripten complier into low level Javascript which can be interpreted by a browser. The COLLADA file importer was reading in files using a Javascript library, and then the Javascript object which reference the data was being used by C++ to build the mesh information in Ranger. All of the information in the Javascript object needed to be cast to C++ types and copied over into std::vector's. The casting, copying, and incremental memory allocation from Javascript to C++ was making the COLLADA importer very slow.

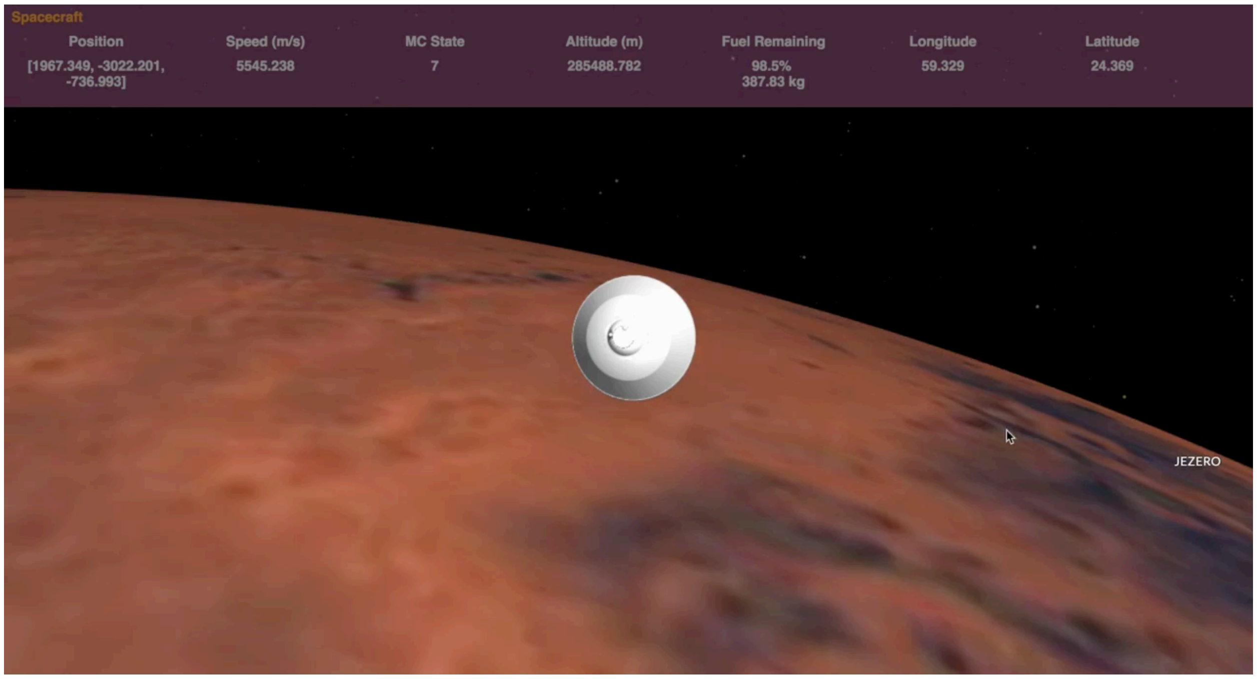
In order to speed up the process, I first hypothesized that building a COLLADA importer in C++ would speed up the process greatly. My thought was that if the data is loaded in C++, it can be flexibly arranged into the format that Ranger required for 3D models without casting and copying. I started by building this COLLADA importer in C++, which took a few days. To parse the XML in C++ (COLLADA is a format built on XML), I used a C++ XML "streaming" library, which meant that the data wasn't loaded into memory by the parser, it was simply scanned through from start to finish. The COLLADA importer in C++ ran extremely fast when compiled natively, and the ISS loading which took a minute before took a few seconds. I then tried to use this new COLLADA importer in C++ within the Ranger system. When integrating the new code, the Ranger COLLADA import still took about a minute to load the ISS. So, I determined that the approach of parsing XML in C++ was not going to work. In hindsight, the native C++ code which ran quickly on my machine was being compiled by Emscripten into Javascript, and so the code was not optimized in terms of memory allocation or reading XML and still ran slowly.

After the C++ approach did not work, I realized that the fastest COLLADA importer would be a library already written for parsing XML in Javascript, because the code has been optimized. The problem was still getting the data into std::vector's that C++ could use. My second approach was to allocate memory in C++ and then have calls to Javascript which read in the data and filled it into the memory allocated by C++. In order to do this, I used the Emscripten memory view object to pass Javascript a pointer to memory allocated in C++. The Javascript code loaded in the COLLADA file in an optimized fashion, and then filled in the memory with the data. When running this code on the ISS import, the code took about 6 seconds, which was a success. The optimization was then merged into Ranger's production code a few days later for a release.

Name	**Speedup**
GPM	19.36x
Aqua	5.04x
Aura	4.50x
GRACE-FO1	3.25x
GRACE-FO2	2.76x
ISS	13.54x
Jason-3	4.37x
OCO-2	5.15x
OSTM	4.81x
SMAP	4.30x
Suomi NPP	3.81x
Terra	13.34

Part 2: Entry, Descent, Landing Visualization

The Ranger browser-based visualization engine has been used by many NASA apps to visualize data. Ranger is being applied to the M2020 rover mission in order to analyze simulation data visually and display real-time telemetry data. The rover will broadcast its telemetry data during the landing sequence, which is fed through a processing pipeline that drives the visualization.



Spacecraft

Position	Speed (m/s)	MC State	Altitude (m)	Fuel Remaining	Longitude	Latitude
[1967.349, -3022.201, -736.993]	5545.238	7	285488.782	98.5% 387.83 kg	59.329	24.369

Here is an image of the Entry, Descent, Landing app built with Ranger. This image displays incoming telemetry data: position, speed, landing state, altitude, fuel, longitude, and latitude.

I contributed to the EDL M2020 application by using Ranger to configure the correct scene for the simulation. I started by loading in all models needed by the simulation (rover folded, rover deployed, parachute, etc.) and configuring the states so that different models would be displayed at different stages of landing. The data processing pipeline we used is called FLOW, and it is a library written in Python for managing telemetry streams on a network.

Once the scene was configured, there were some difficulties in verifying the visualization's accuracy that were worked out later. For instance, the translation and rotation of the spacecraft models was not an easy thing to check. In addition, the rotation of Mars was an issue complicated by the system's time reference point and the frame of reference in which the rotation occurred. In the end, the simulation's scene was adjusted and its accuracy was verified.