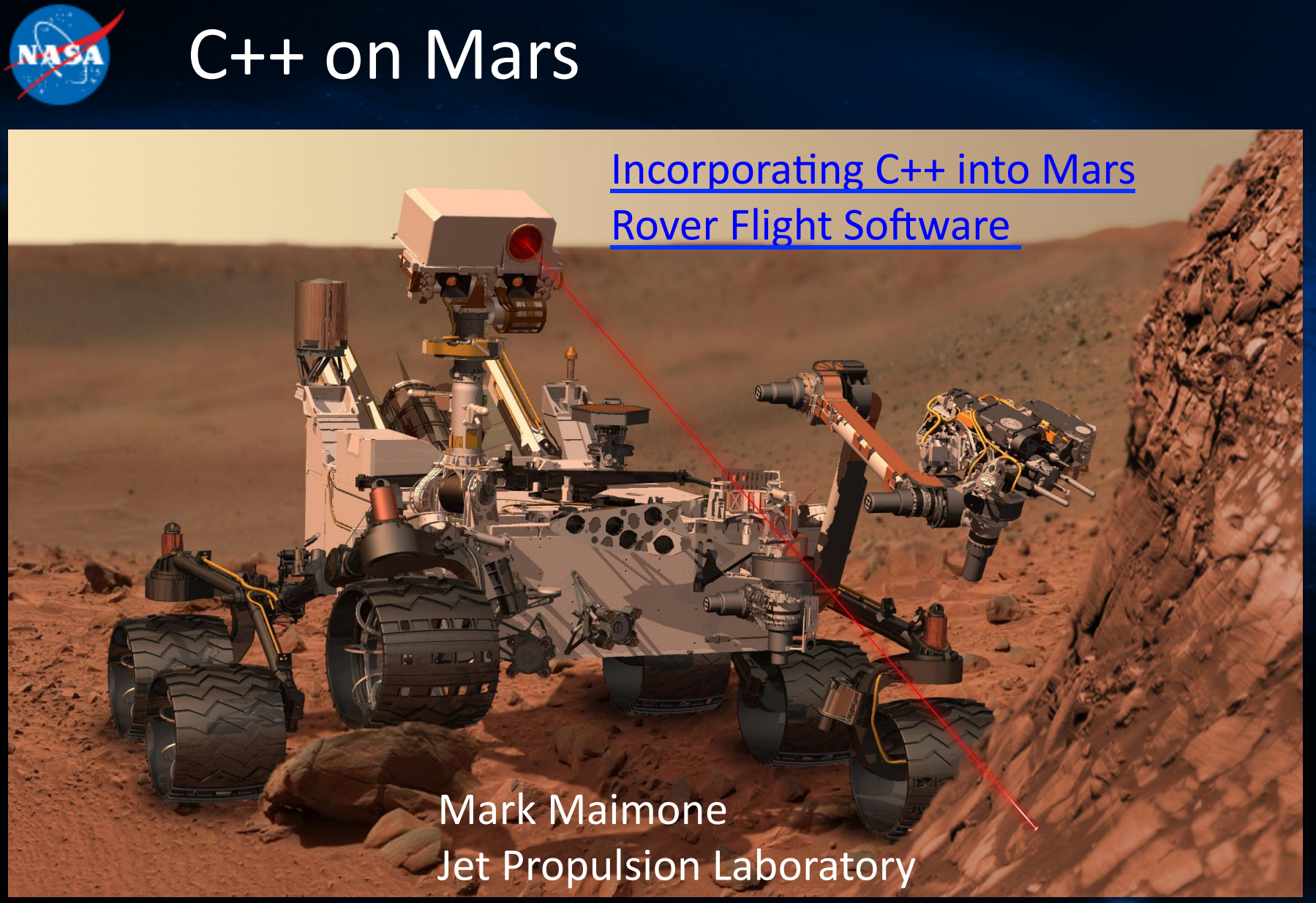


C++ on Mars

Incorporating C++ into Mars Rover Flight Software

Mark Maimone
Jet Propulsion Laboratory
California Institute of Technology

Artist's Concept. NASA/JPL-Caltech

A Journey Through Time and Space

Imagine …

A Journey Through Time and Space

Imagine …

Y2K

A Journey Through Time and Space

Imagine …

Y2K

A little Embedded Systems Project

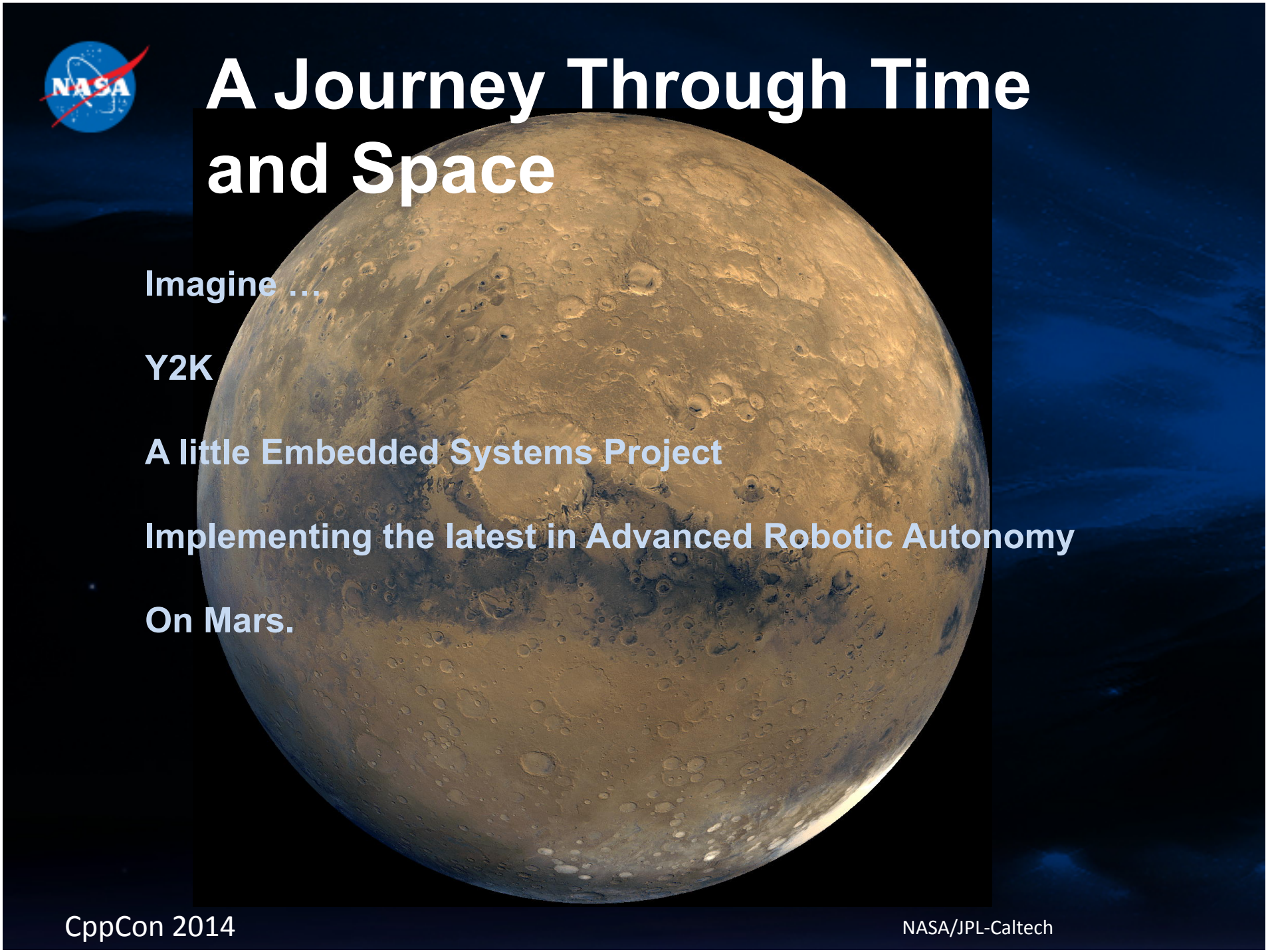
A Journey Through Time and Space

Imagine …

Y2K

A little Embedded Systems Project

Implementing the latest in Advanced Robotic Autonomy



A Journey Through Time and Space

Imagine …

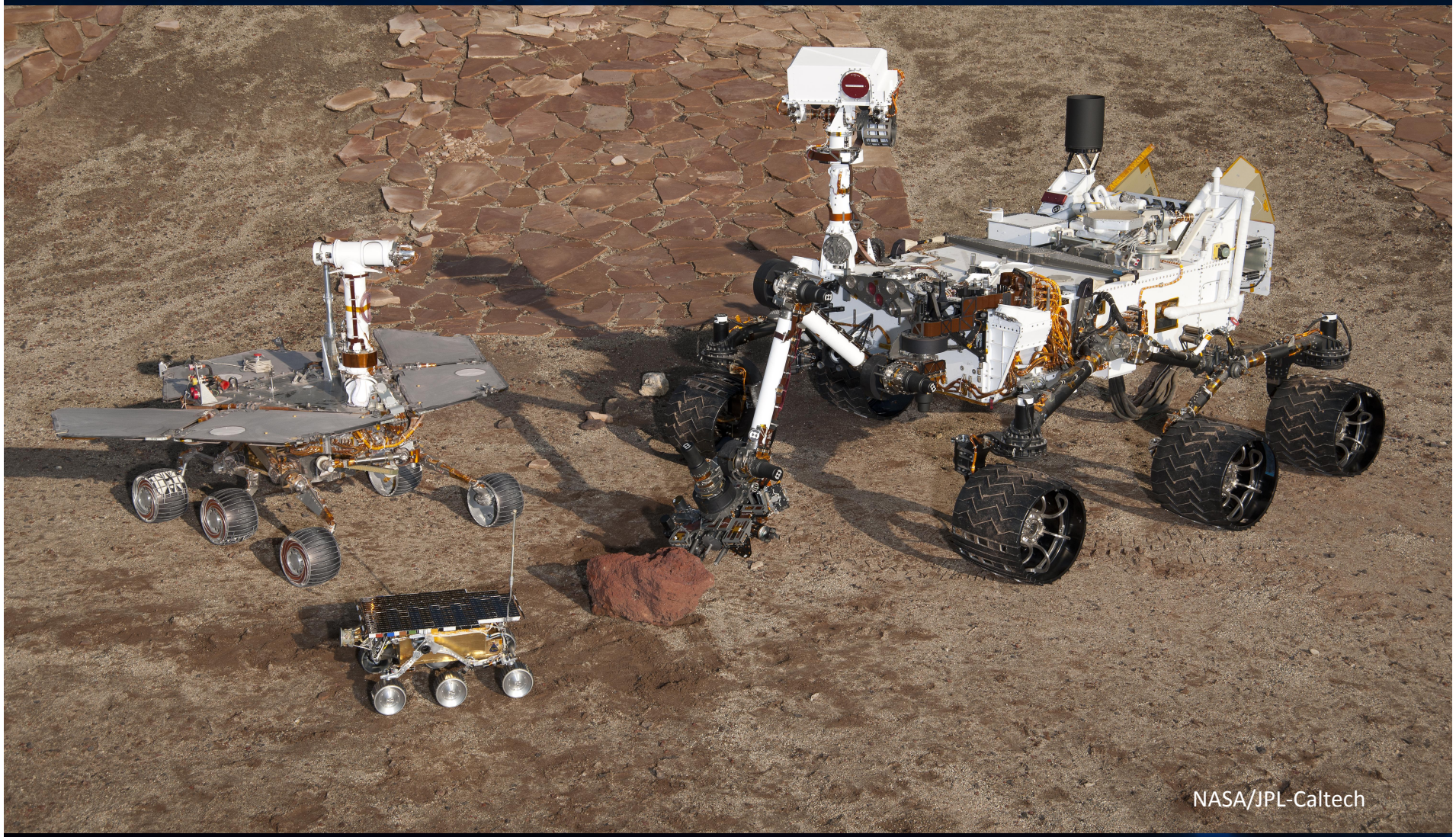
Y2K

A little Embedded Systems Project

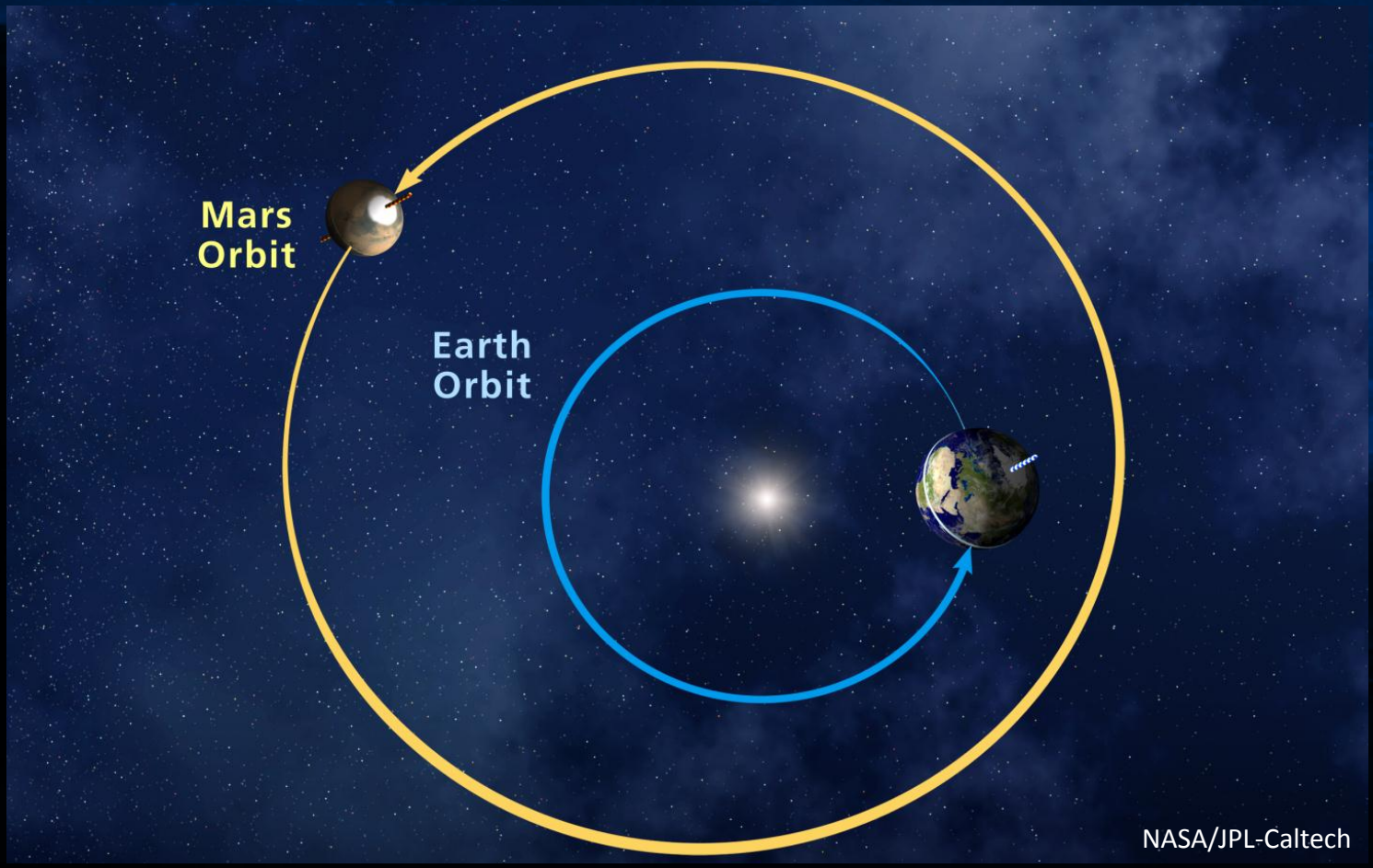
Implementing the latest in Advanced Robotic Autonomy

On Mars.

Meet NASA's Mars Rovers



Because of the distance between Earth and Mars, we can't drive a rover in real time.



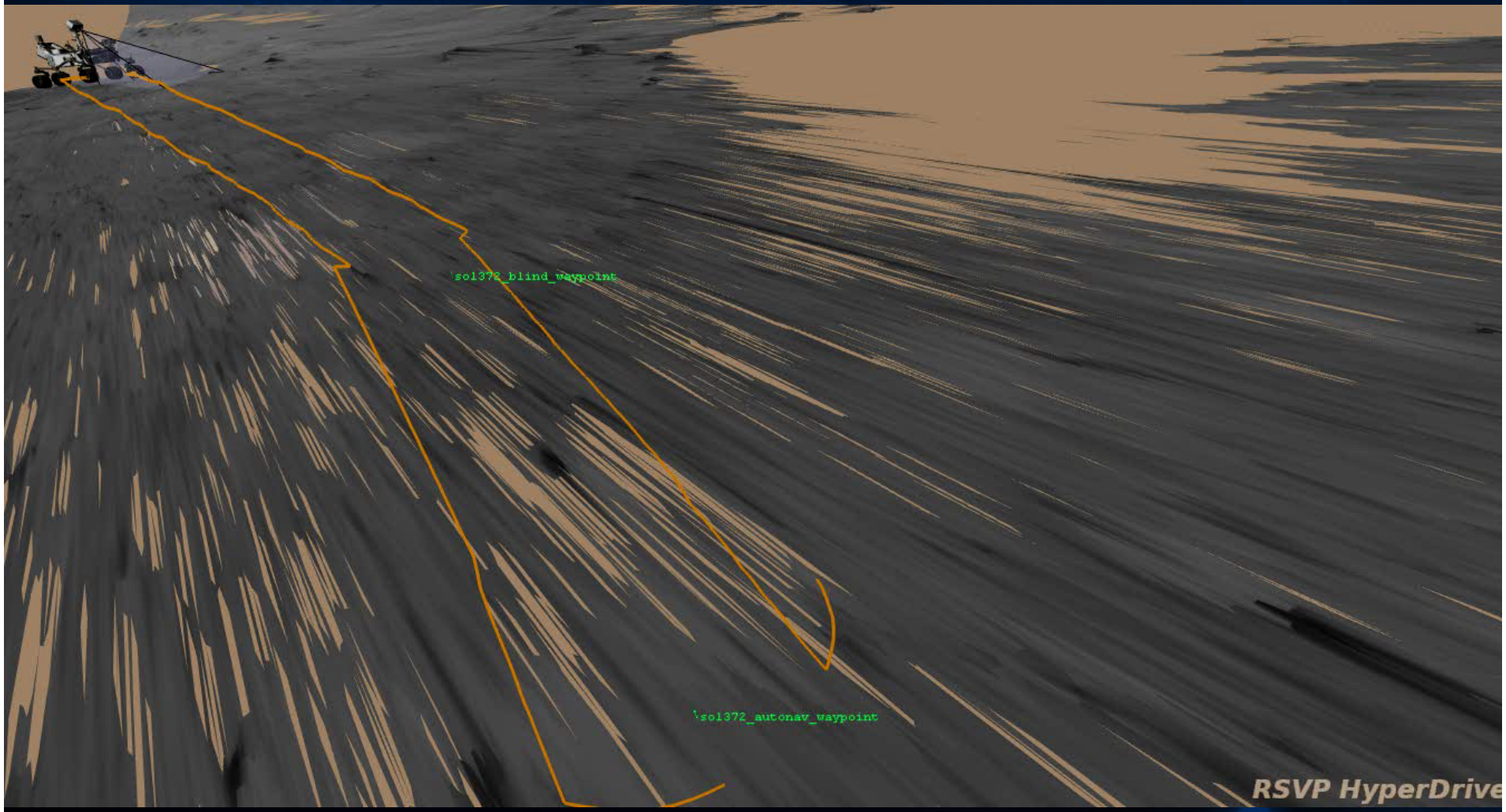
It takes between 4 and 22 minutes each way for a signal to travel between the two planets.

Also, Logistics

The Deep Space Network is a shared resource for dozens of missions.

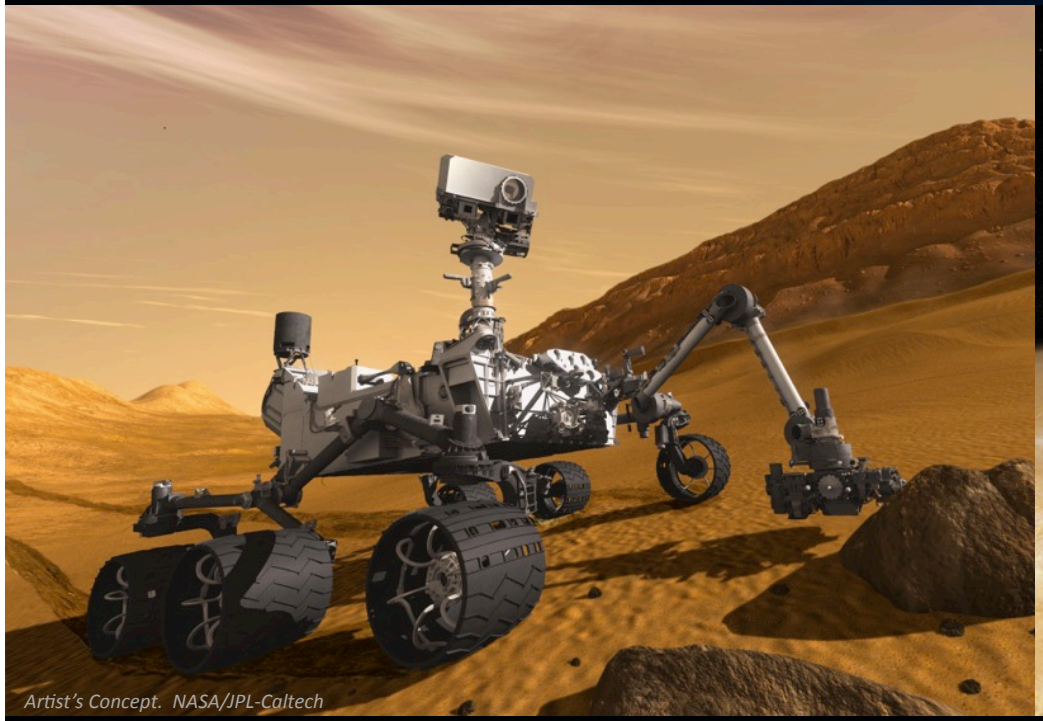
We often only get one uplink and few downlink windows each day

A Day's Plan Might Include Driving, Using the Arm, or Remote Science



NASA/JPL-Caltech

Mars Rovers carry out their activities then send data to the orbiters, whose larger antennas relay it to Earth.



Artist's Concept. NASA/JPL-Caltech

Artist's Concept. NASA/JPL-Caltech

It takes less energy and a smaller antenna to send data 200 miles (322 km) up to an orbiter, rather than millions of miles to Earth, though direct contact is available.

So How Do You Build One?

Start with robust, fault tolerant hardware and software designs

Squeeze in as much autonomous capability as you can get approved

Foreshadowing: C++!

Test as you fly. Test again. Test. Test. Test.

Software Resources

	MER	MSL
Radiation-hardened CPU	RAD6000 (PowerPC)	RAD750 (PowerPC)
Clock Speed	20 MHz	133 MHz
On-board RAM	128 Mbytes	128 Mbytes
Real Time Operating Sys	VxWorks 5.3.1	VxWorks 6.7
Addressable Code RAM	32 Mbytes	32 Mbytes
FSW + RTOS Code Size	10 Mbytes	21 Mbytes
Additional RAM	n/a	512 Mbytes SDRAM (half for RAMFS)
Per-Task Memory access	Shared Memory	Shared Memory
C/Embedded C++ compiler	Green Hills MULTI 3.5	GCC 4.1.2

Mars Rover FSW Modules

MER and MSL each have over 100 individual source code modules

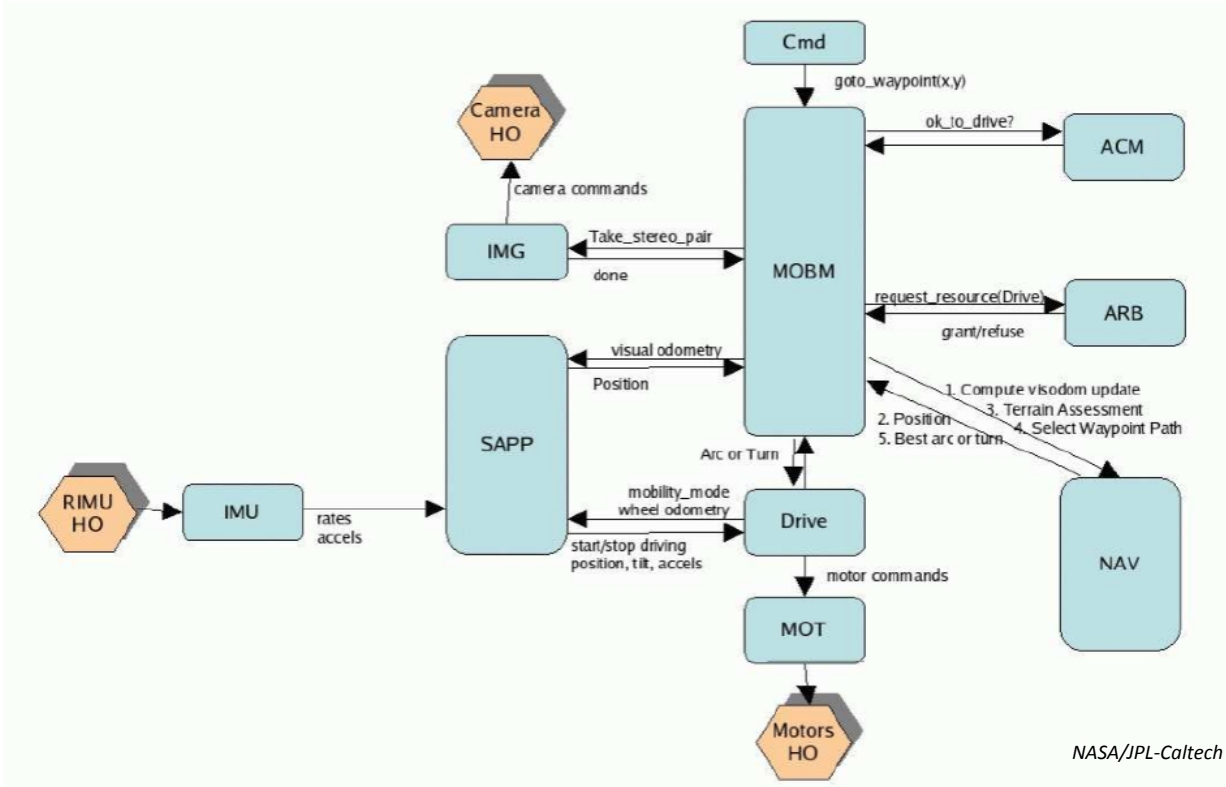
- Each module had a single FSW developer as its owner

Modules typically communicate by passing messages through Message Queues assigned to each task

- Very minimal use of semaphore locks, shared memory

Learn more:
"*Curiosity's FSW Architecture: A Platform for Mobility and Science*", Dr. Kathryn Weiss, NASA JPL, video from 2012 Flight Software Workshop

MER Mobility FSW Modules



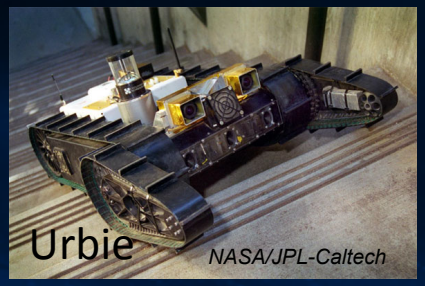
Surface navigation is the most complex module, comprising 21% of MER object code, 10% of MSL. How to manage its complexity?

Use C++ !

Most of the onboard autonomous driving software on MER and MSL is written in C++

Dense Stereo Vision
Autonomous Terrain Assessment
Local and Global Waypoint Planning
Multi-sol Driving
Visual Odometry
Slip Checks
Keepout Zone Prediction

Why C++?



Throughout the 1990's, JPL researchers used C++ to develop high level autonomous behaviors like stereo vision, map building, path planning and visual odometry.

C++ class abstraction and encapsulation enabled rapid development and testing among multiple projects and developers . **Many capabilities were field-tested and field-proven over years of testing.**

Why was C++ So Late to Space?

Before MER development began in 2000, C++ code had not flown on any JPL Mars mission.

Spaceflight projects always want to minimize risk, so prefer software environments with flight heritage.

So we weighed the risk of using the new environment against the risk of rewriting a mature and field tested existing C++ codebase.

C++ won!

What Does our C++ Code Do?

Here is an overview of the Mars Rover C++ software.

Most of it is based around the automatic interpretation of stereo pairs of images taken by the rover as it moves across the surface of Mars.

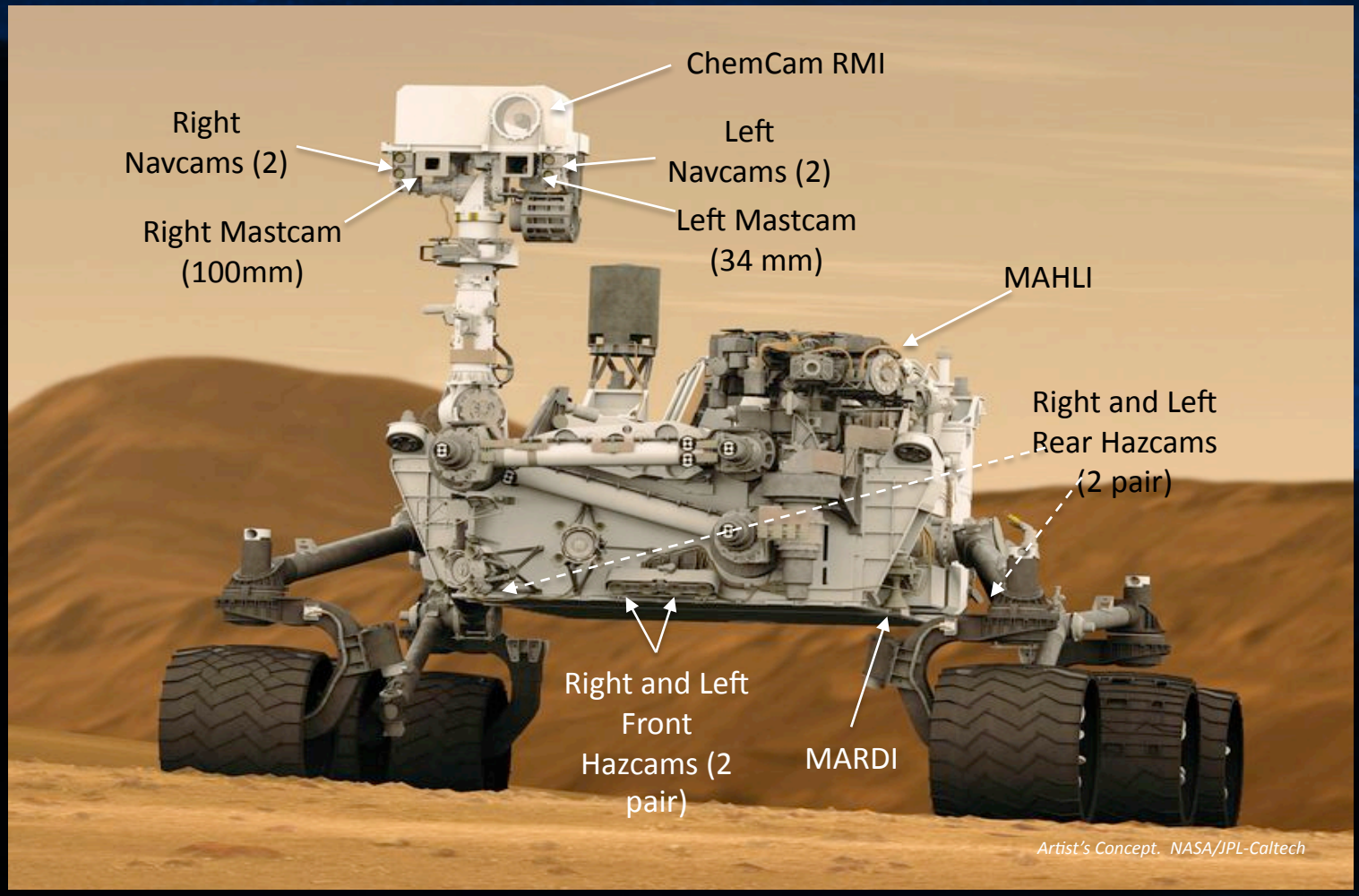
Next: cameras, Autonomous driving, and Visual Odometry

Learn more: *Leave the Driving to Autonav,* Curiosity Rover Report, Sept. 19, 2013

Two Years of Visual Odometry on the Mars Exploration Rovers, Maimone, Cheng, Matthies, JFR Vol 24 no 3, 3/2007, pp 169-186.

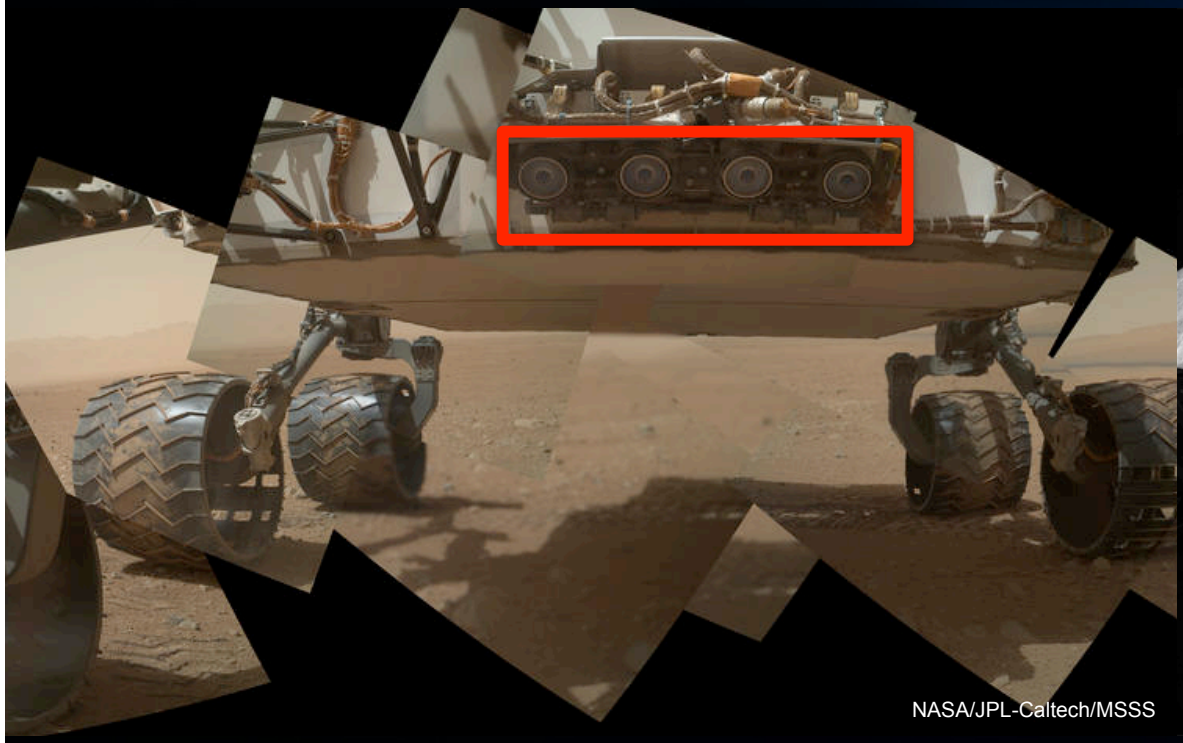
The Mars Exploration Rover Surface Mobility Flight Software: Driving Ambition, Biesidecki, Maimone, IEEE Aerospace 3/2006.

Curiosity has 17 cameras

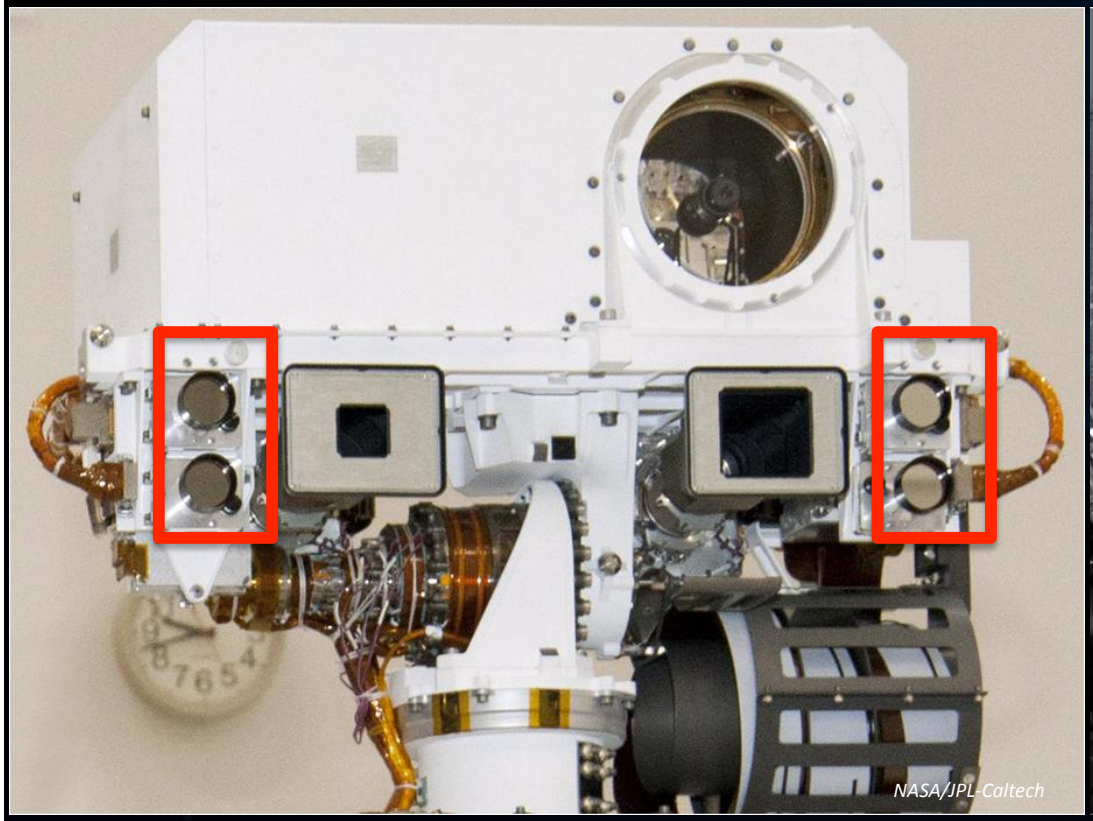


Hazcams and Navcams are tied into the auto-nav software.

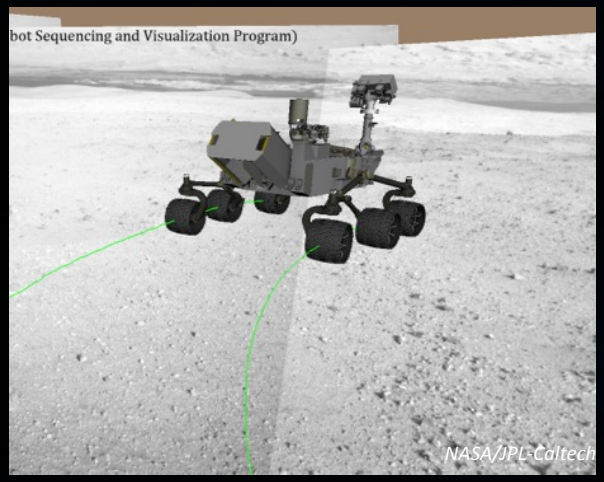
The hazard avoidance cameras give a 120° wide angle view of the area near the rover. Front cameras have 16cm baseline, rear cameras have 10cm baseline.

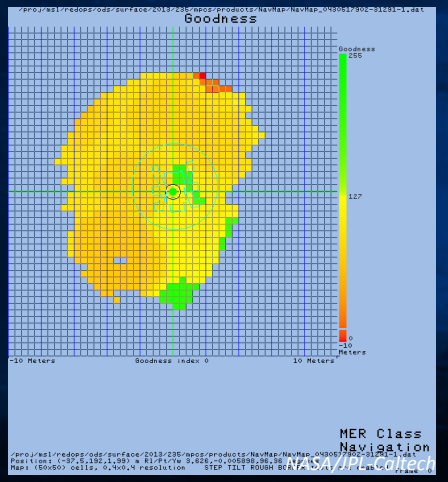


The 45° navigation cameras are almost 7 feet off the ground with 42cm baseline, providing good views over nearby obstacles or hills and into ditches.



Human Rover Drivers Decide How Much Autonomy is Desired Based on Terrain and Available Resources



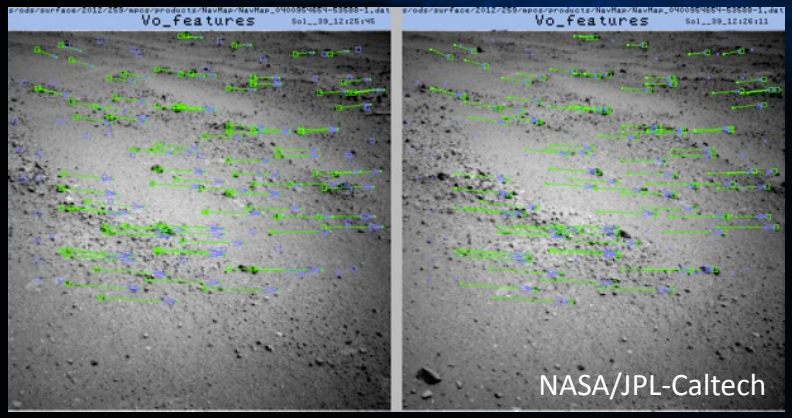


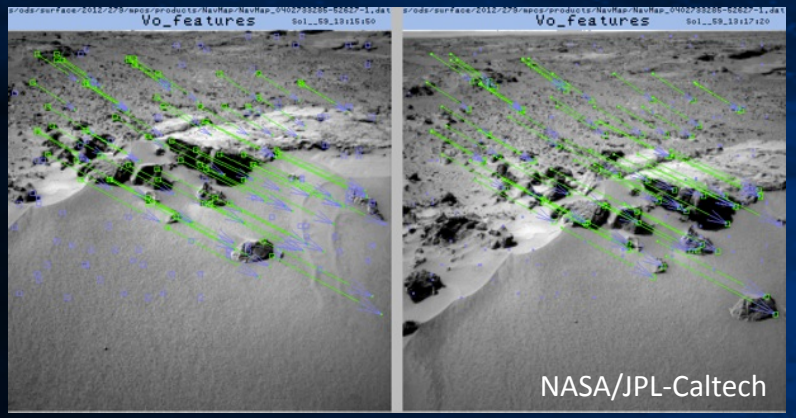
Directed driving

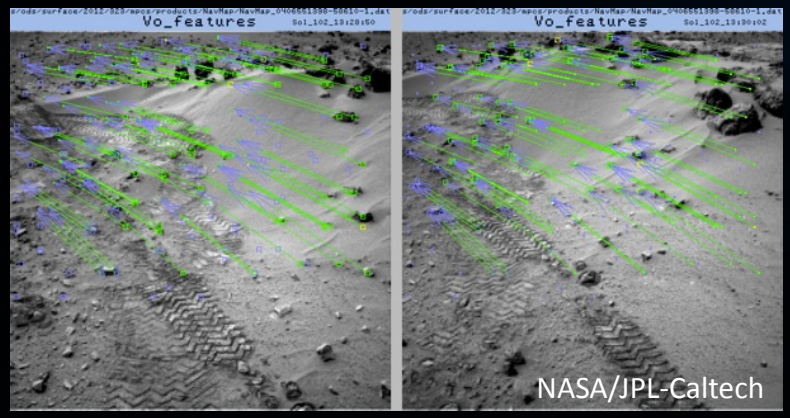
Visual odometry, or
Slip Check + “Auto”

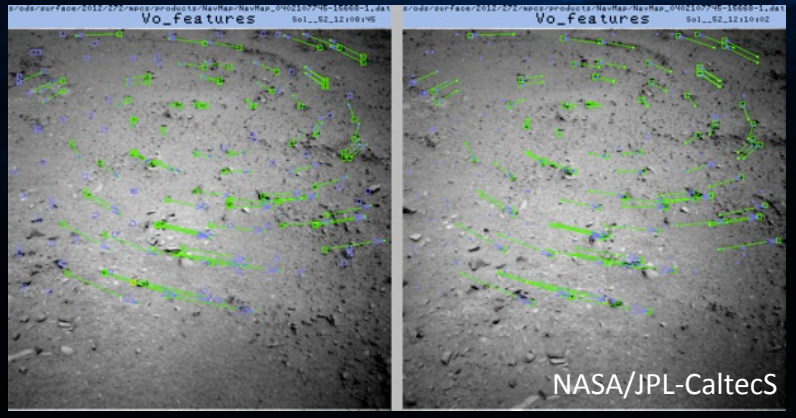
Auto-navigation;
Geometric Hazard
Detection and
Avoidance

Using visual odometry, the rover constantly compares pairs of images of nearby terrain to calculate its position.



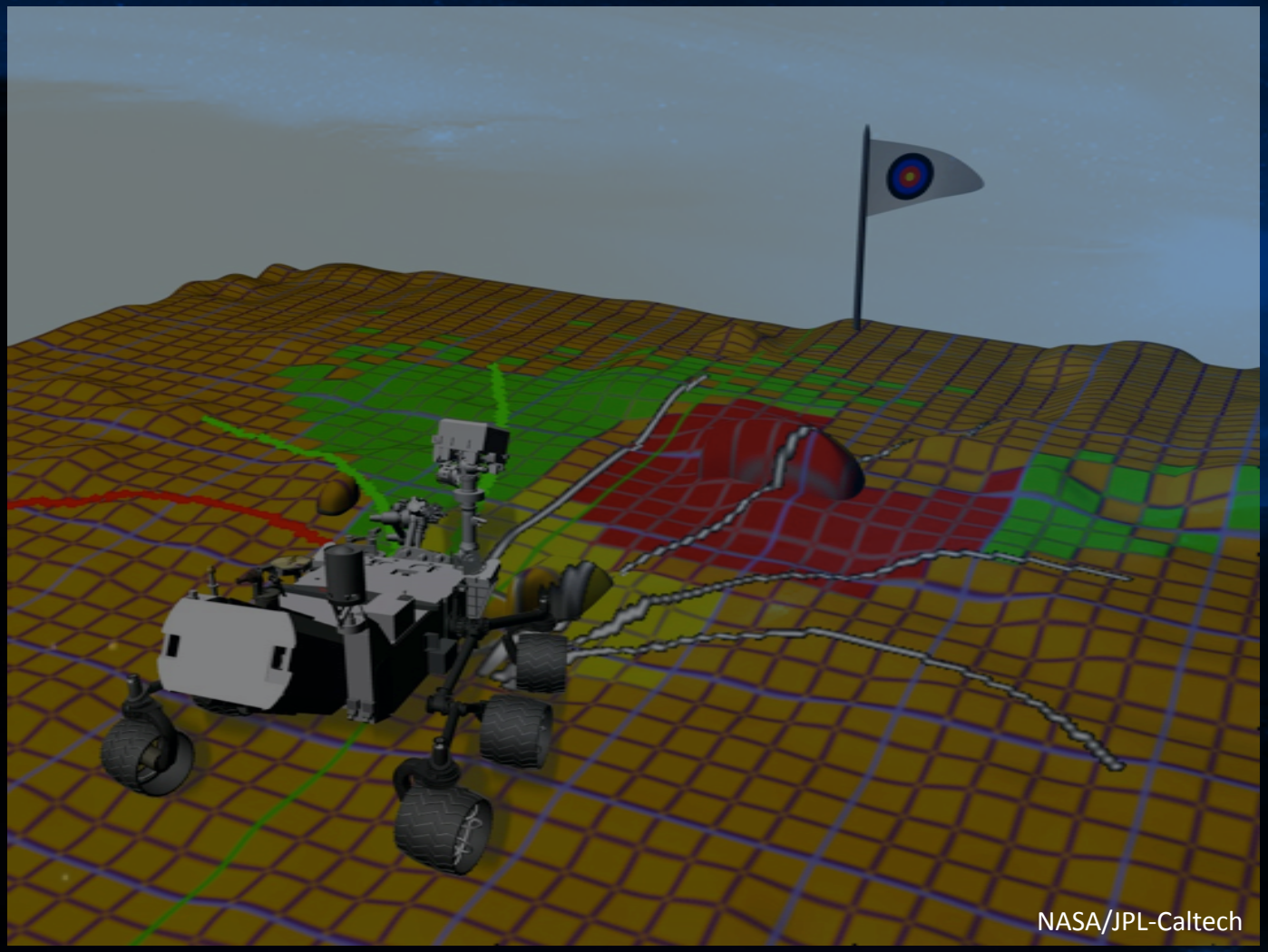






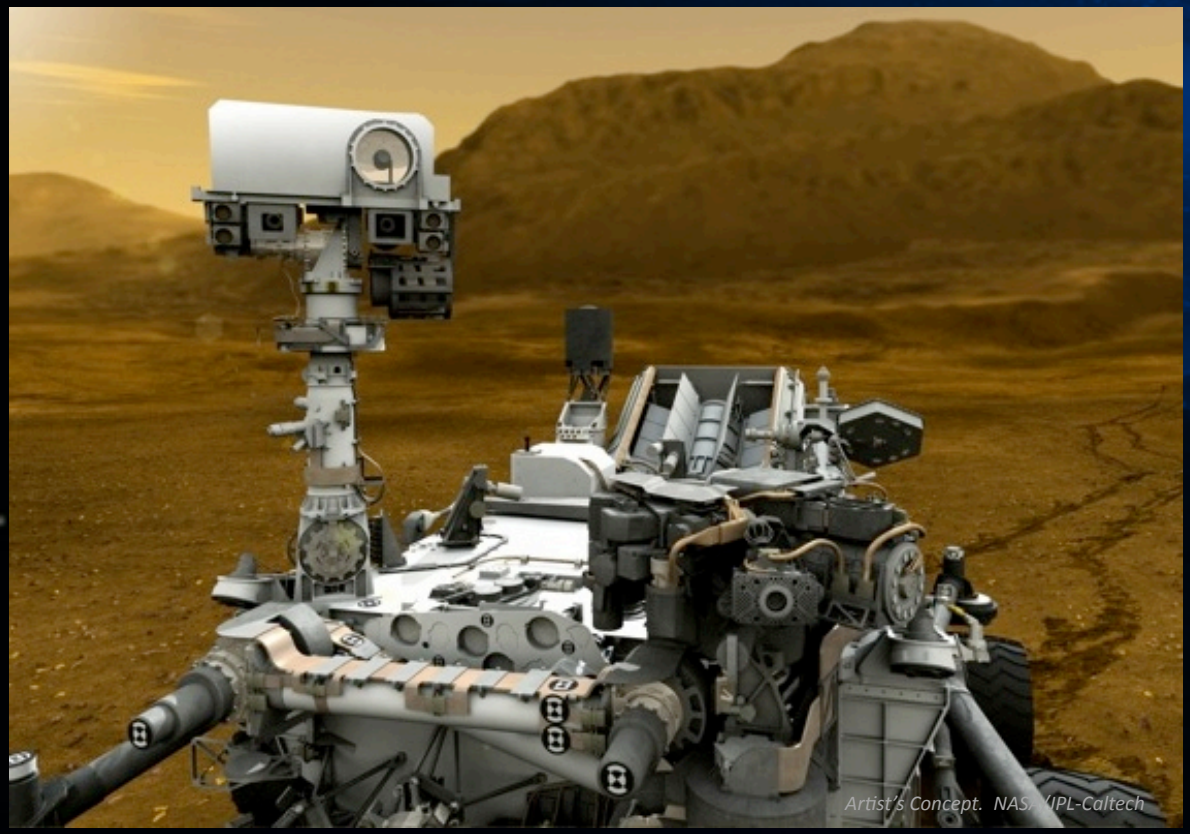
Unlike terrestrial robots, Curiosity drives as far as possible between VO images

Rover Navigation 101



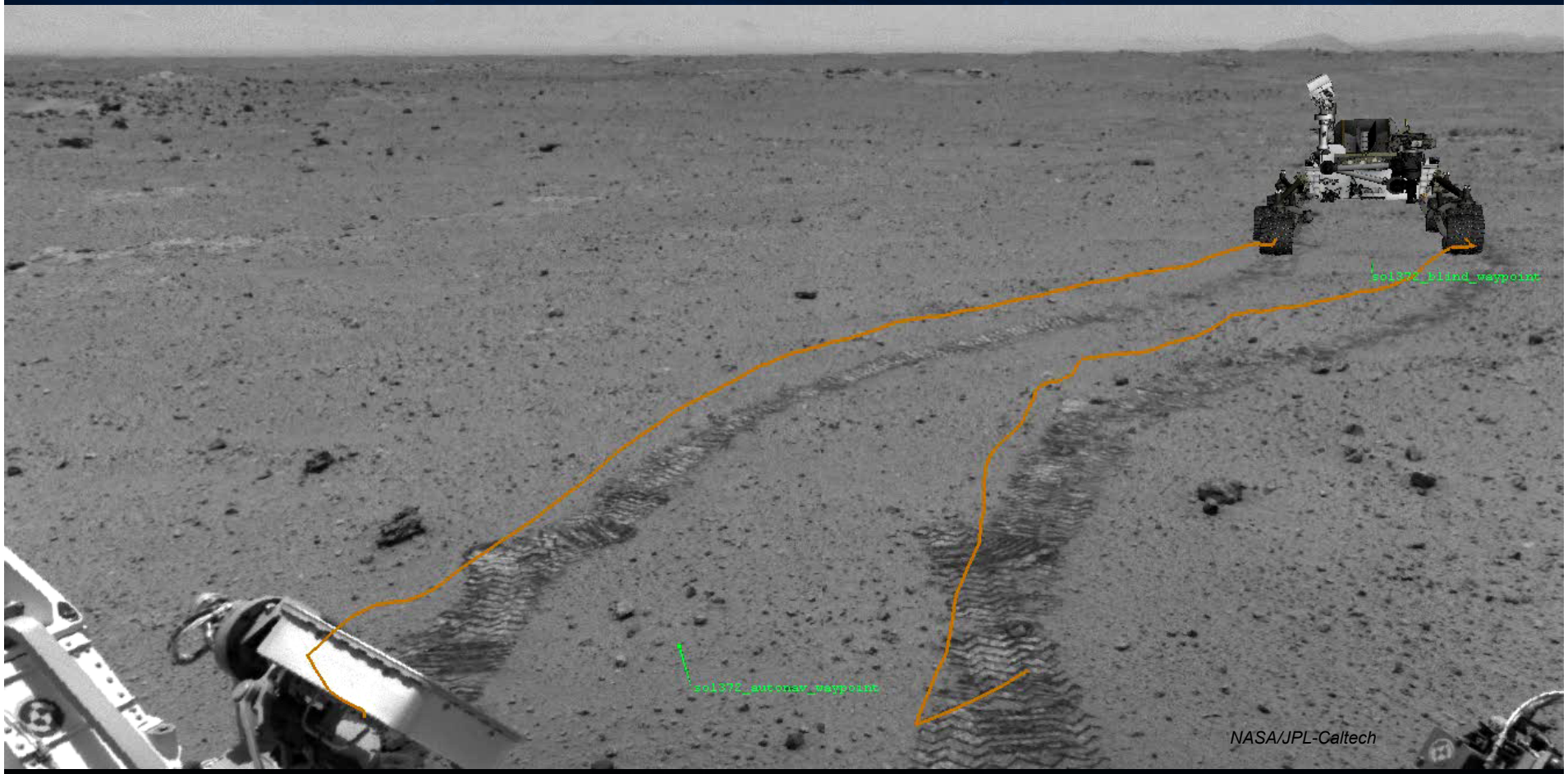
Learn More: *Rover Navigation 101*

To drive around hazards, the rover stops every 0.5-1.5 meters, takes 4 sets of images, evaluates hazards, and then chooses where to drive.



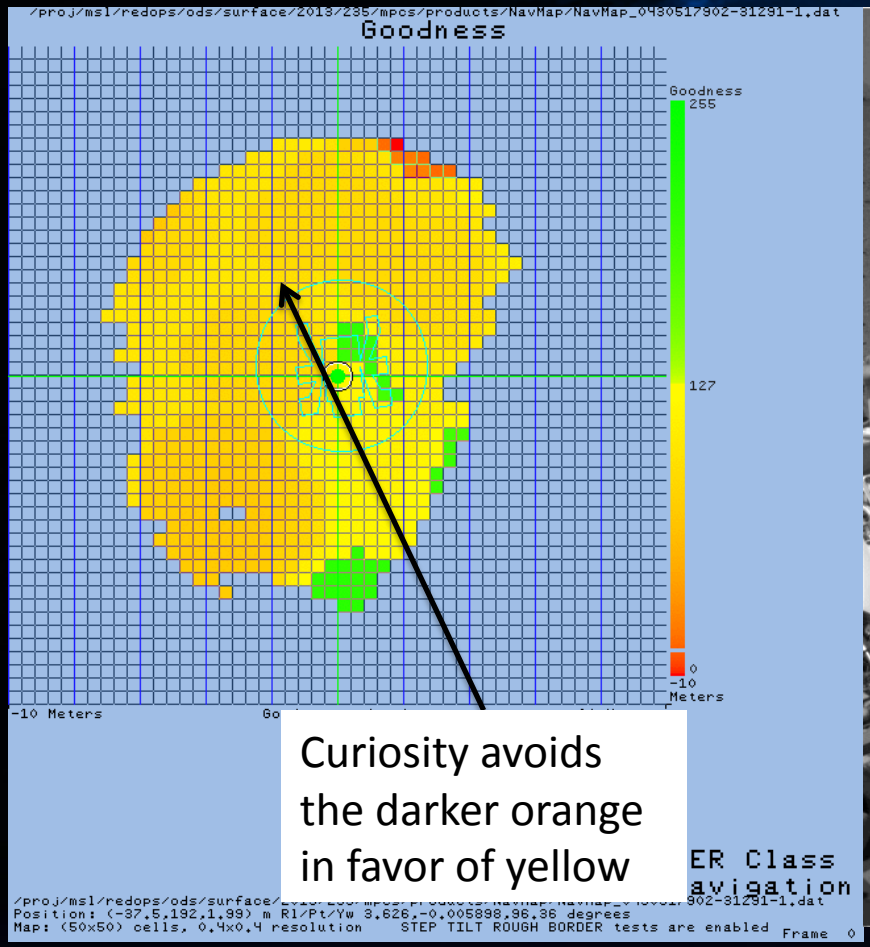
Auto-nav extends directed drives into previously unseen terrain

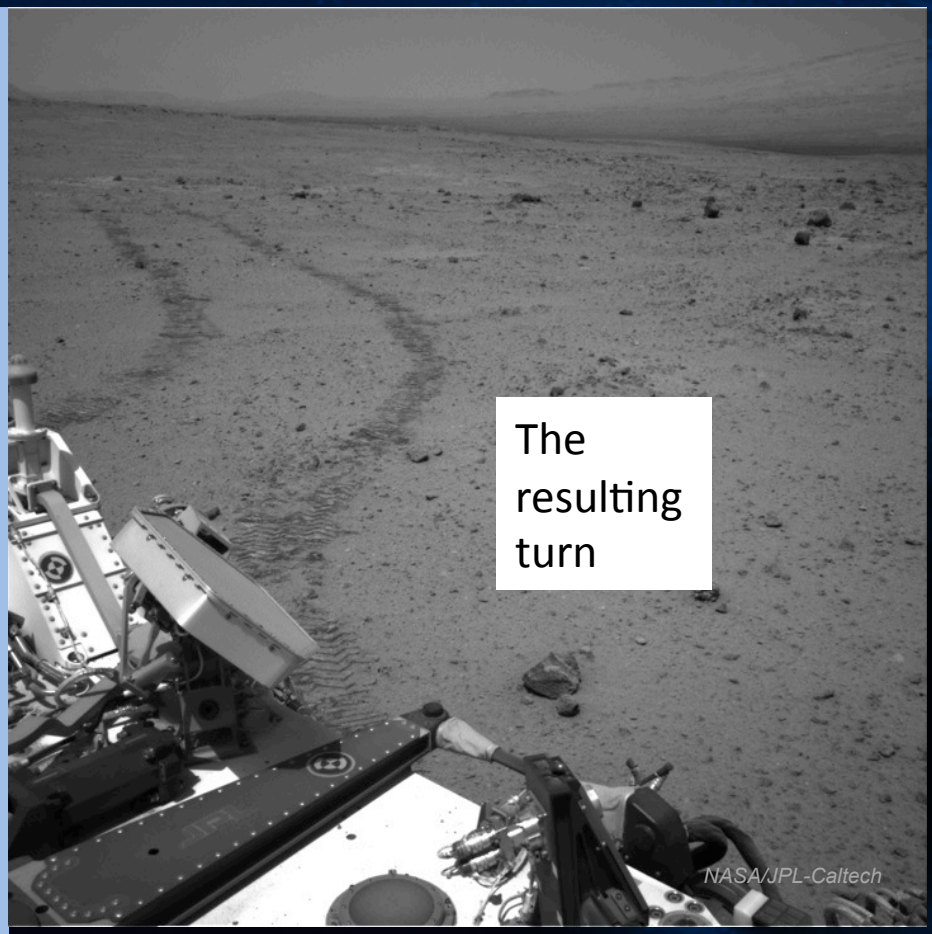
Animation of Curiosity's actual Sol 372 drive over a picture of her tracks



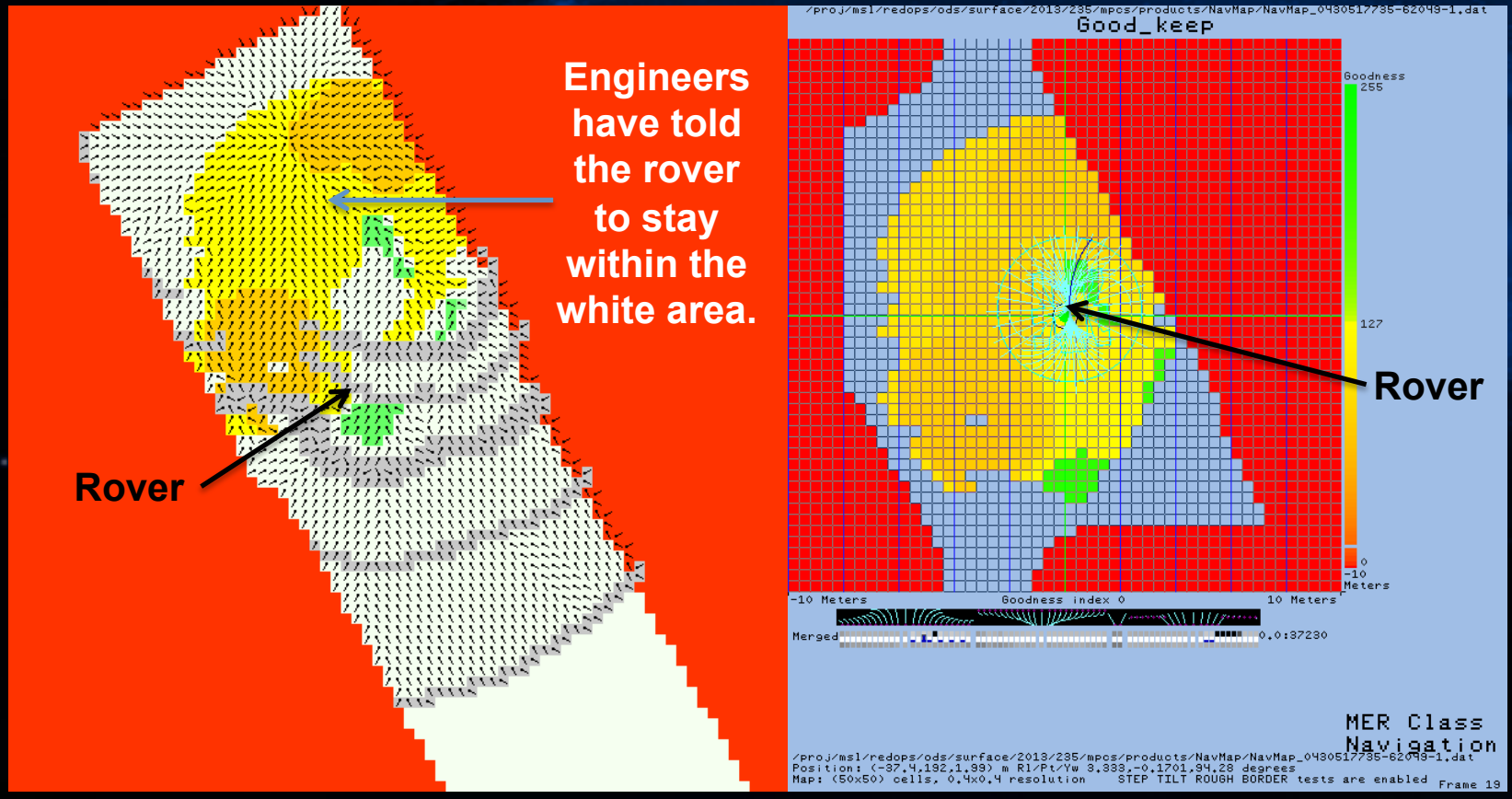
Finish!

Curiosity's map and tracks show this decision to turn was based on her evaluation of the terrain.





The rover reduces a stereo point cloud into a configuration space, labeling unsafe areas red and safe areas green.



NASA/JPL-Caltech

Yellow means drive carefully, just like on Earth.

MER FSW Updated in 2006

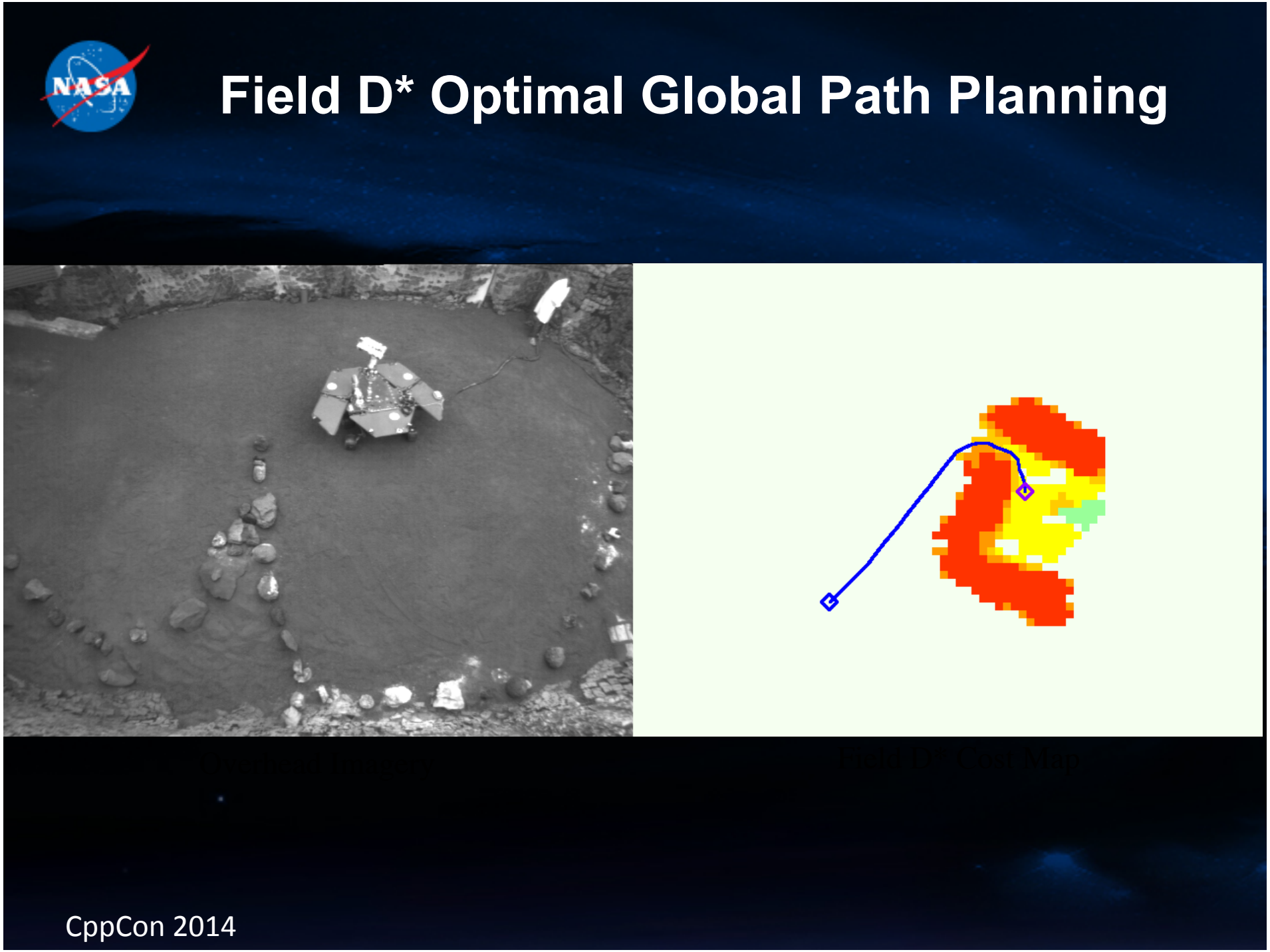
R9.2 of the MER FSW included several then-new technologies, several using C++

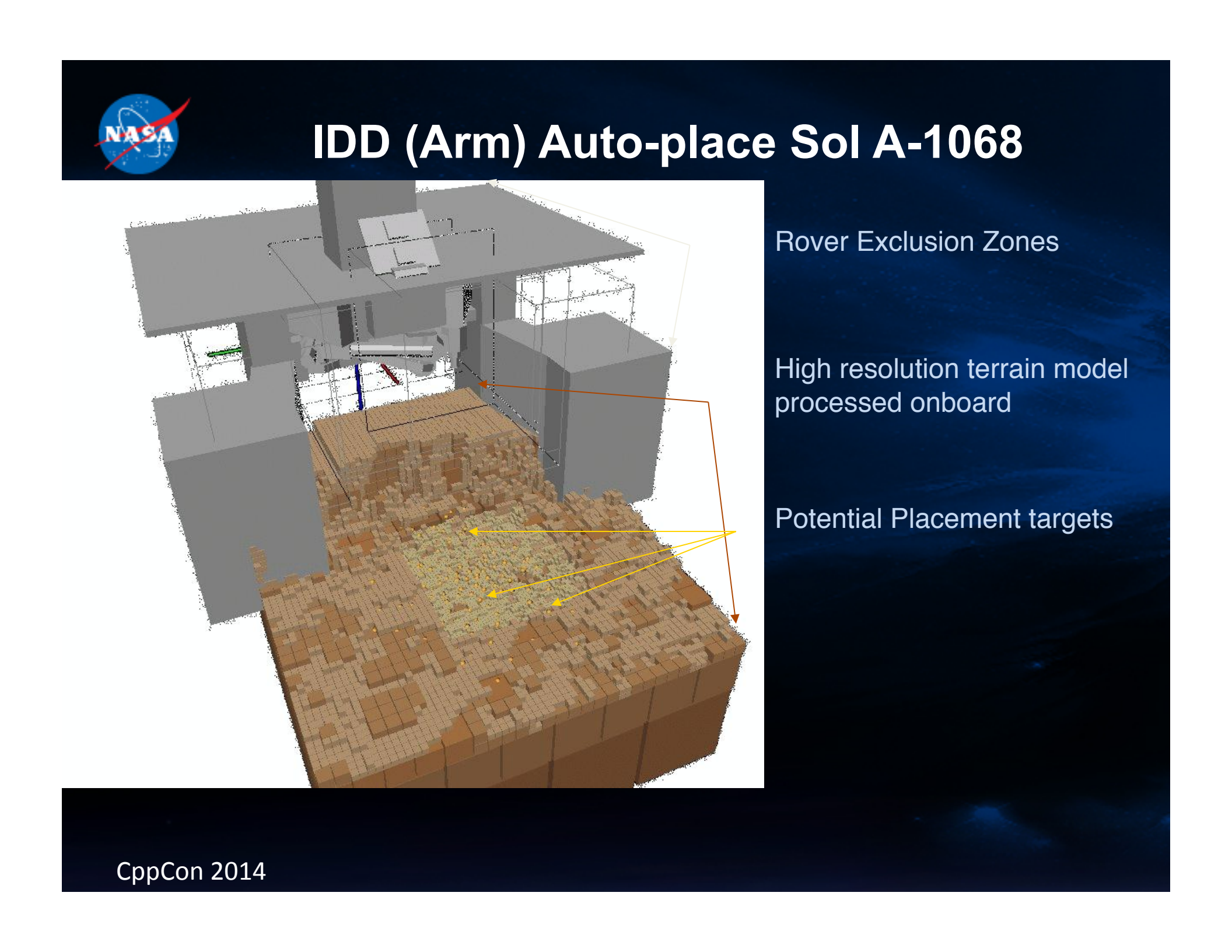
Field D* - Optimal Long Range Drive Planning, now standard in MSL

IDD Autoplace (Go and Touch) – Place the arm on a target autonomously after driving toward it

Visual Target Tracking – Fast tracking of a nearby terrain feature (new version now being checked out on MSL)

Learn more: *Overview of the Mars Exploration Rovers' Autonomous Mobility and Vision Capabilities*, Maimone, Leger, Biesiadecki, ICRA 2007.


Field D* Optimal Global Path Planning
Overhead Imagery
Field D* Cost Map
CppCon 2014

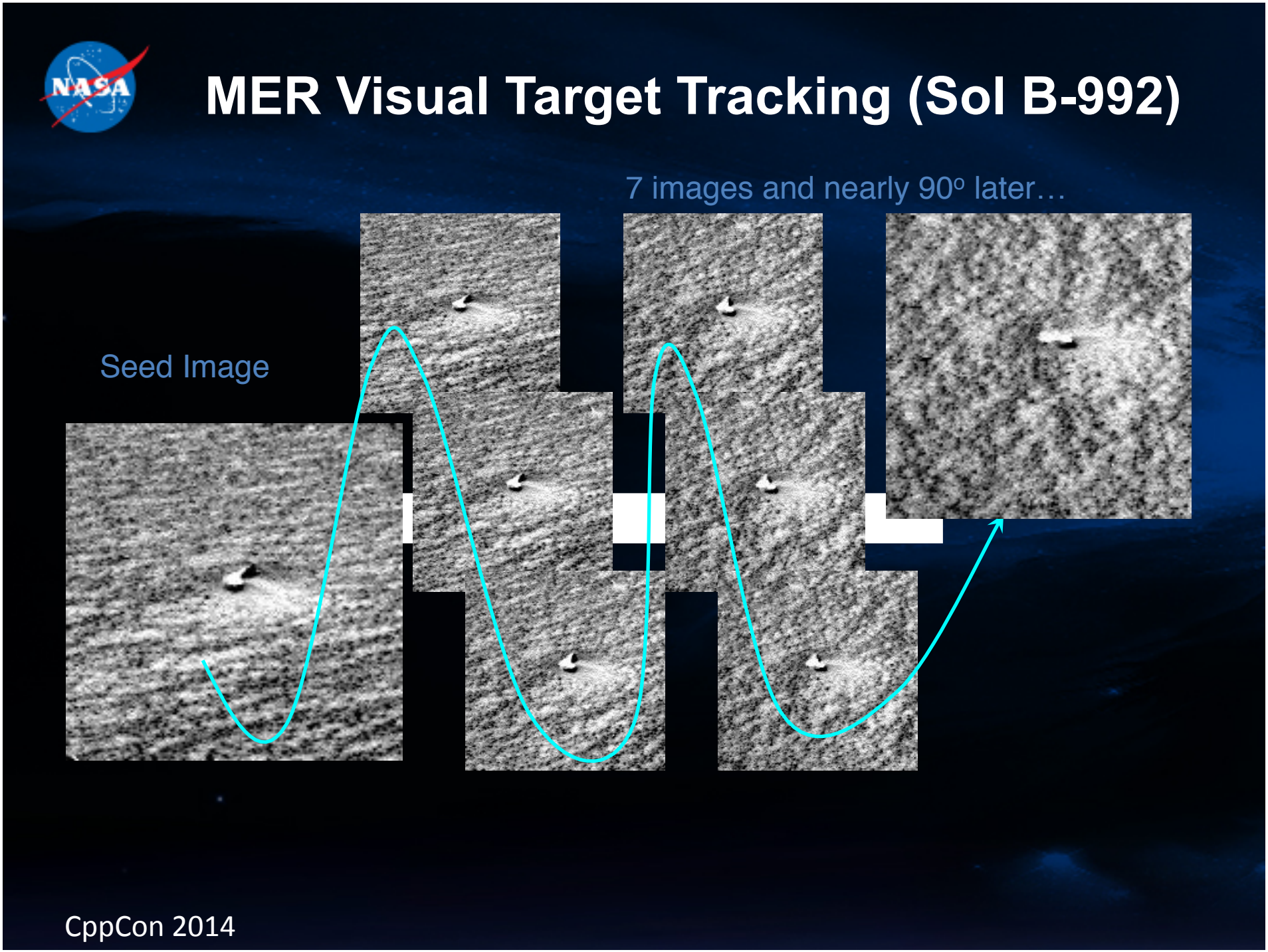


IDD (Arm) Auto-place Sol A-1068

Rover Exclusion Zones

High resolution terrain model processed onboard

Potential Placement targets


MER Visual Target Tracking (Sol B-992)
7 images and nearly 90º later…
Seed Image
CppCon 2014

Initial Concerns about C++

Historic arguments against using C++ in flight included:

Exceptions: Too much uncertainty, difficult to validate all possible control paths

Templates: Easy to cause code bloat

Iostream: Console output deprecated when your console is 200 million miles away

Multiple inheritance: little experience in our environment

Operator overloading: confusing for other developers

Dynamic allocation: worries about running out of system heap RAM, uncertainty of timing during garbage collection

Using Embedded C++

So we limited our code to “Embedded C++” constructs

Exceptions: none

Templates: none

Iostream: none

Multiple inheritance: none

Operator overloading: almost none (only “new” and “delete”).

Dynamic Allocation: Guarantee no system heap corruption using a dedicated memory pool and Placement New.

Placement New

Recall that our O/S uses shared memory across all tasks; every task shares the same system heap for dynamic allocations.

So how to guarantee that our new C++ code would not interfere with spacecraft operations?

Overload “new” and “delete” operators to invoke our own memory allocator, never calling the O/S supplied functions

Never touch the system heap: use Placement New syntax to “place” new allocations into explicit RAM addresses inside our separate memory pools.

Memory Allocator

We developed our own Memory Manager

Guarantees graceful access to defined memory pools
- Well-defined behavior for out of memory conditions

Supports multiple pools in different areas of RAM

Provides diagnostics
- Optional display of each new and delete operation
- Maintains free space map available for documentation

Same allocator used for VxWorks and unix development
- Detailed memory tests could run without full testbed

No garbage collector; leaks must be eliminated (enforced during unit tests)

Using the Memory Allocator

Code practices initially dictated that dynamic allocation be eliminated, or restricted to one-time-only during the boot up phase

- But this restriction was waived once shown safe

During autonomy development on Unix and unit testing in VxWorks, we use the detailed diagnostics to trace every allocation to prove no leaks.

During operations, we dump the free map after every complex autonomy step to prove no leaks, or provide data if one occurs.

Running Tests

Unit tests: Developers add Unit tests to ensure changes do not break existing capabilities, enable Code Coverage analysis, and run memory checkers (Valgrind, purify).

Static Analysis: Runs a suite of local and commercial tools

Validation & Verification: A separate test team takes delivered code and runs it through its paces in the various testbeds

Always try to keep tests as realistic as feasible: Test As You Fly.

Learn more: *Mars Code*, by Gerard Holtzmann, CACM Vol 57 No 2, pp 64-73, Feb 2014.

MER FSW Simulation Environments

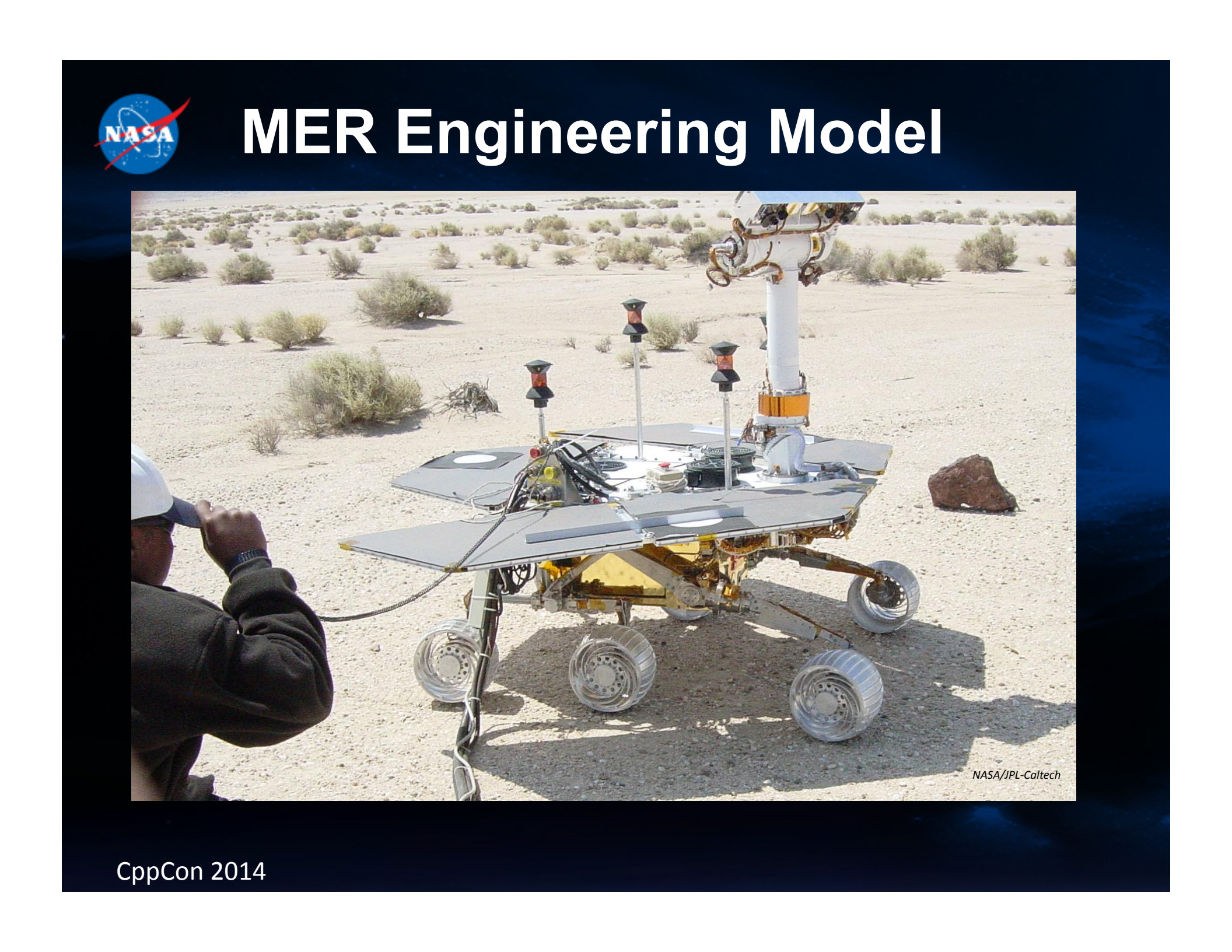
MER team used a variety of testbeds for development

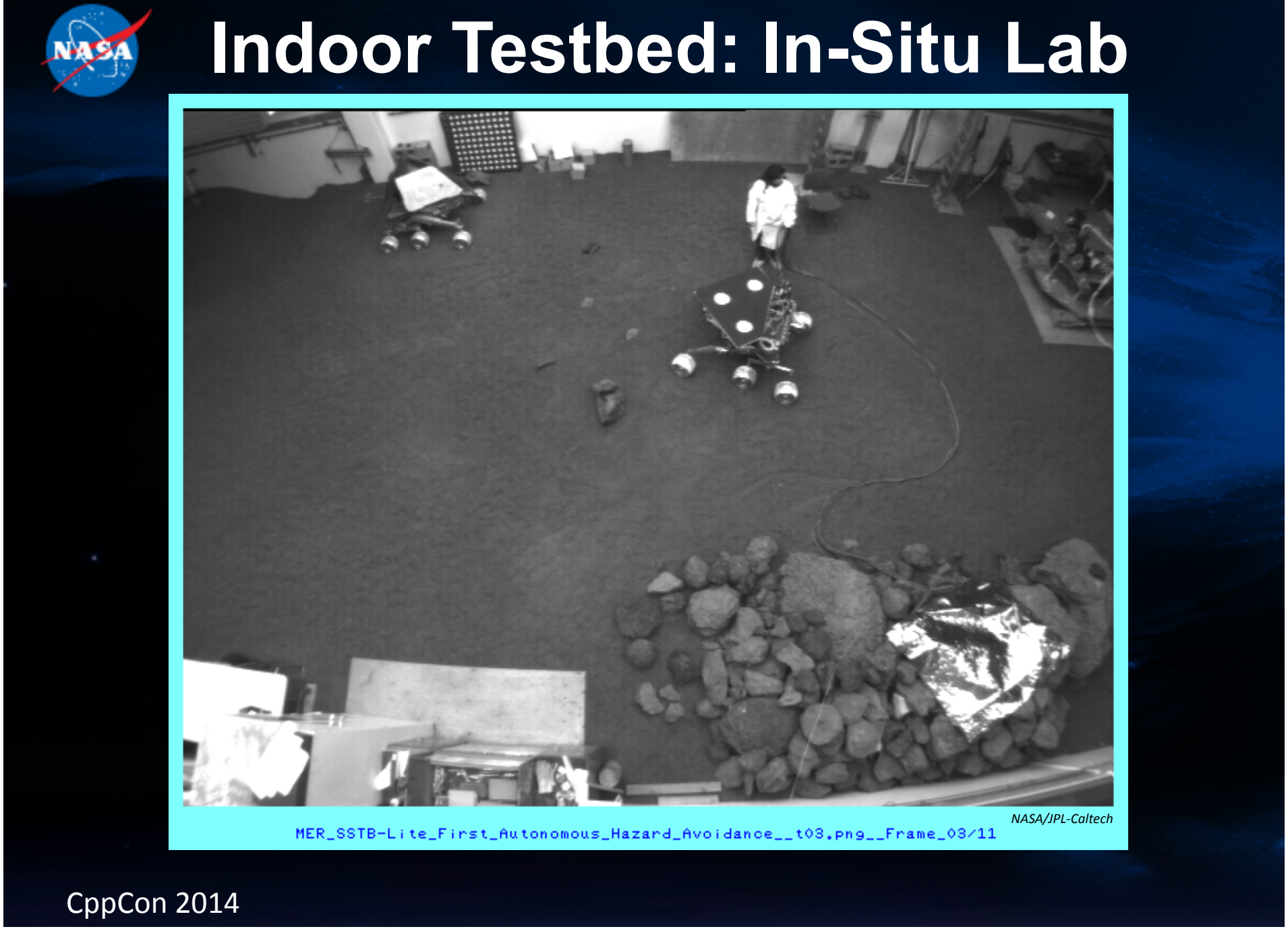
Surface Navigation Unix binary: Just surface navigation library with a dedicated test interface.

Avionics Simulators: dedicated PowerPC boards with software emulation of motors, sensors, filesystems

Testbeds: Flight-like PowerPC boards with flash, EEPROM, and sometimes other hardware in the loop

Surface System Testbed: Full rover Engineering model with sensors, mobility, manipulation, mast


MER Engineering Model
NASA/JPL-Caltech
CppCon 2014



Indoor Testbed: In-Situ Lab

NASA/JPL-Caltech

MER_SSTB-Lite_First_Autonomous_Hazard_Avoidance__t03.png__Frame_03/11

MSL FSW Simulation Environments

Surface Navigation Unix binary

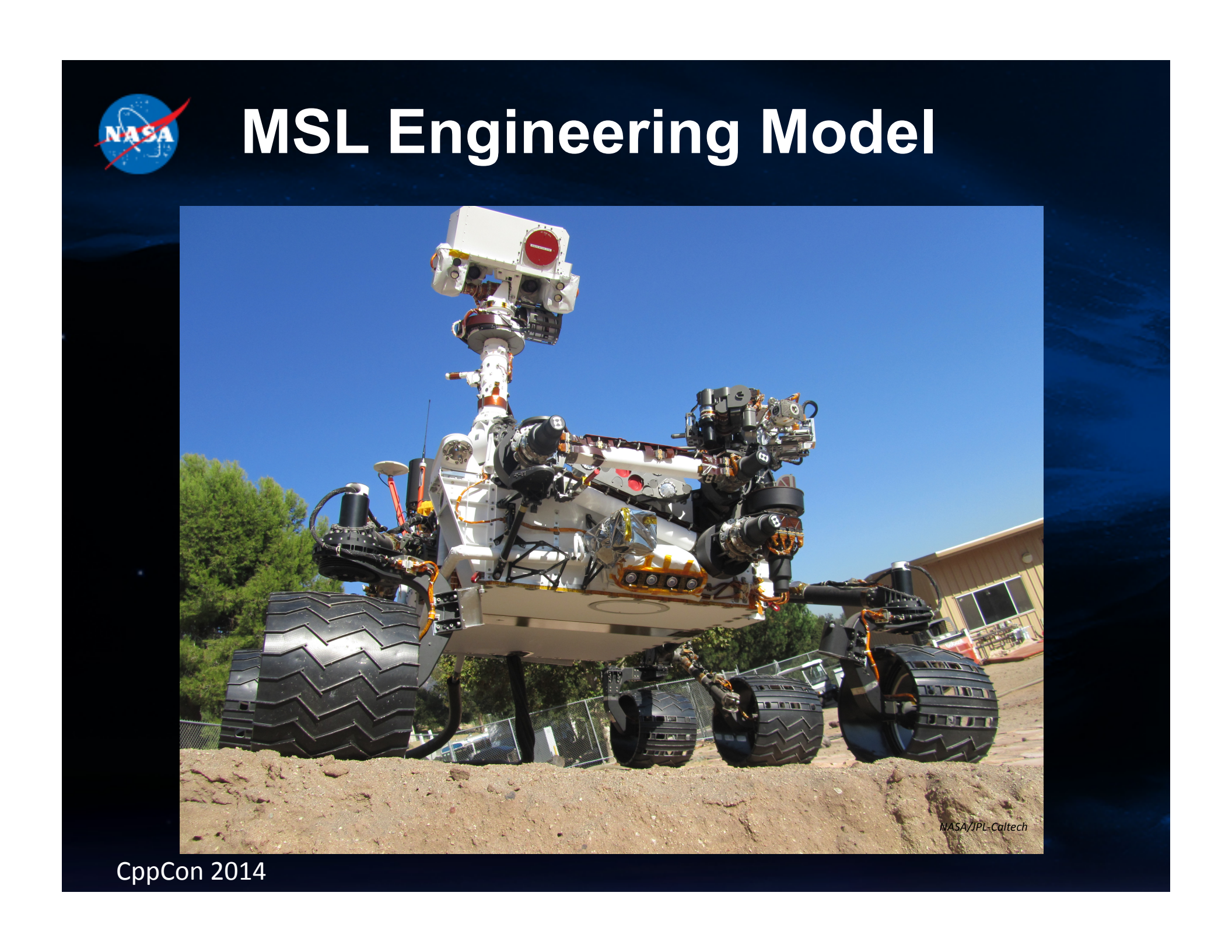
Navsim Unix binary: Software emulation of just mobility spacecraft commands with a 3D terrain renderer

Surface Simulation unix binary (SSIM): Arm and turret command simulation and visualization

Workstation Test Sets (WSTS): VxSim software emulation of flight software, motors, sensors, filesystem, 3D terrain

Testbeds: Flight-like PowerPC boards with flash, EEPROM, and sometimes other hardware in the loop

Vehicle System Testbed (VSTB): Full rover Engineering model with sensors, mobility, manipulation, mast


MSL Engineering Model
NASA/JPL-Caltech
CppCon 2014

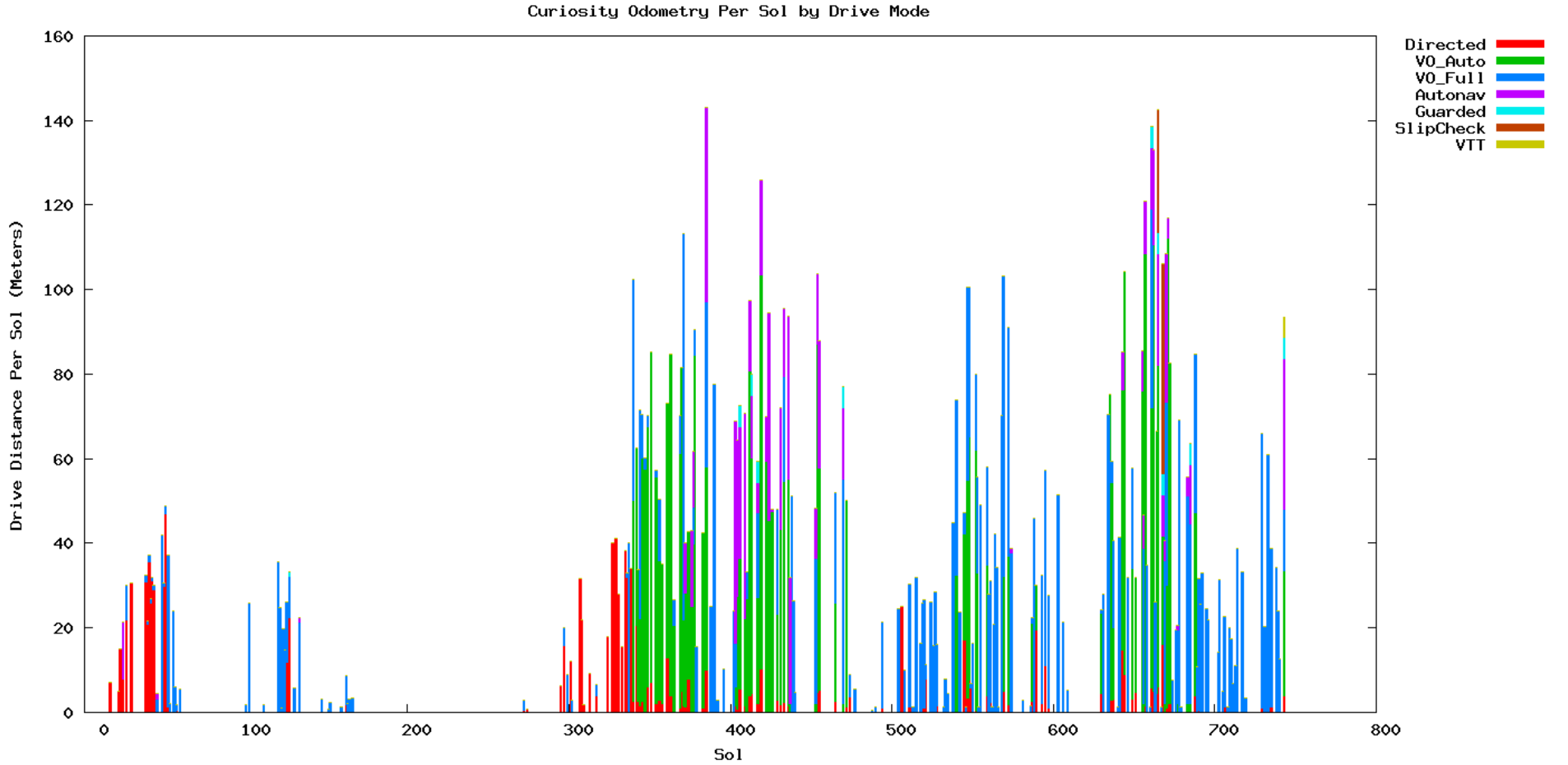
VSTB Driving: 1

NASA/JPL-Caltech

VSTB Driving: 2

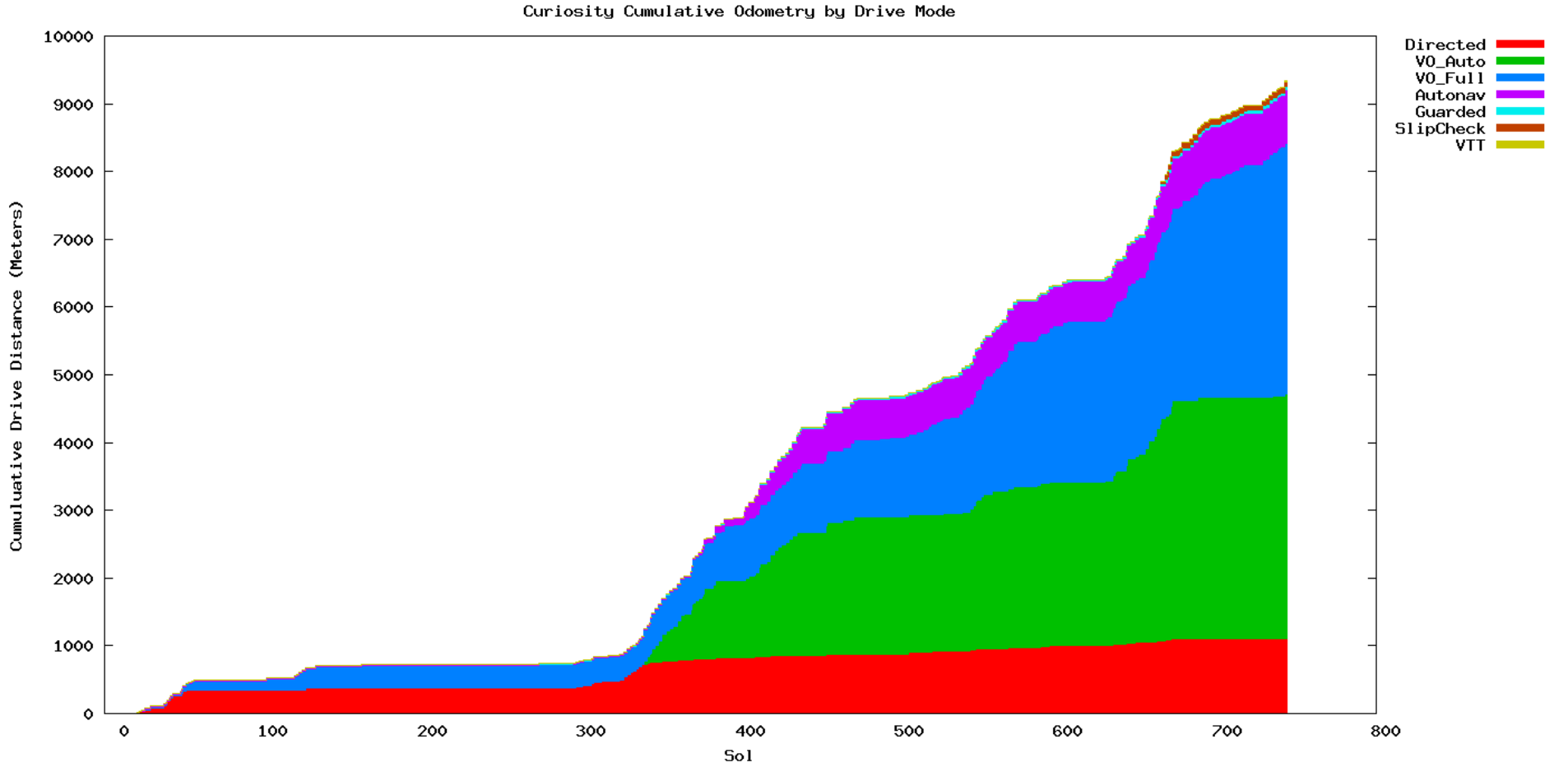
NASA/JPL-Caltech

Curiosity Odometry Per Sol

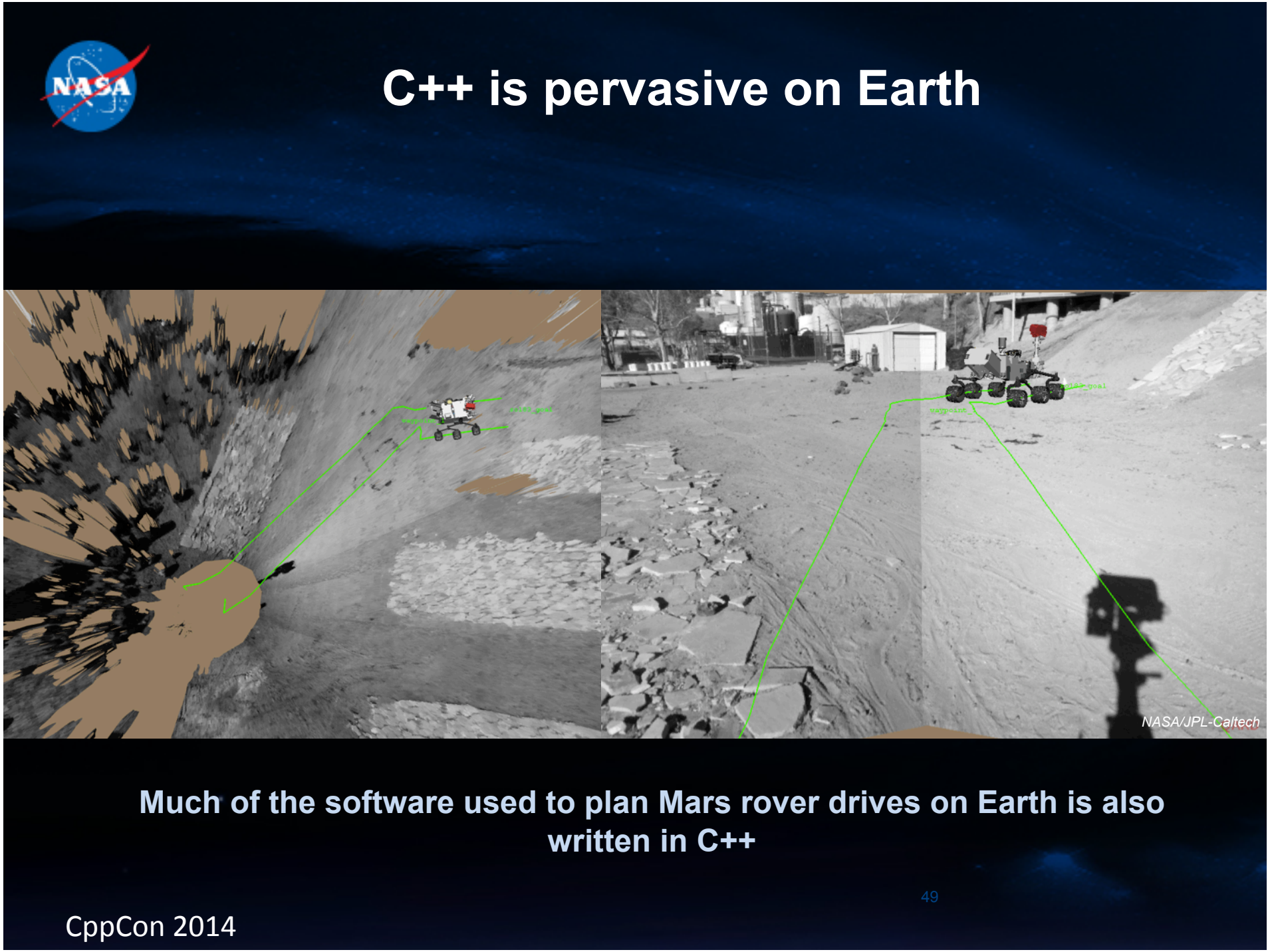


NASA/JPL-Caltech

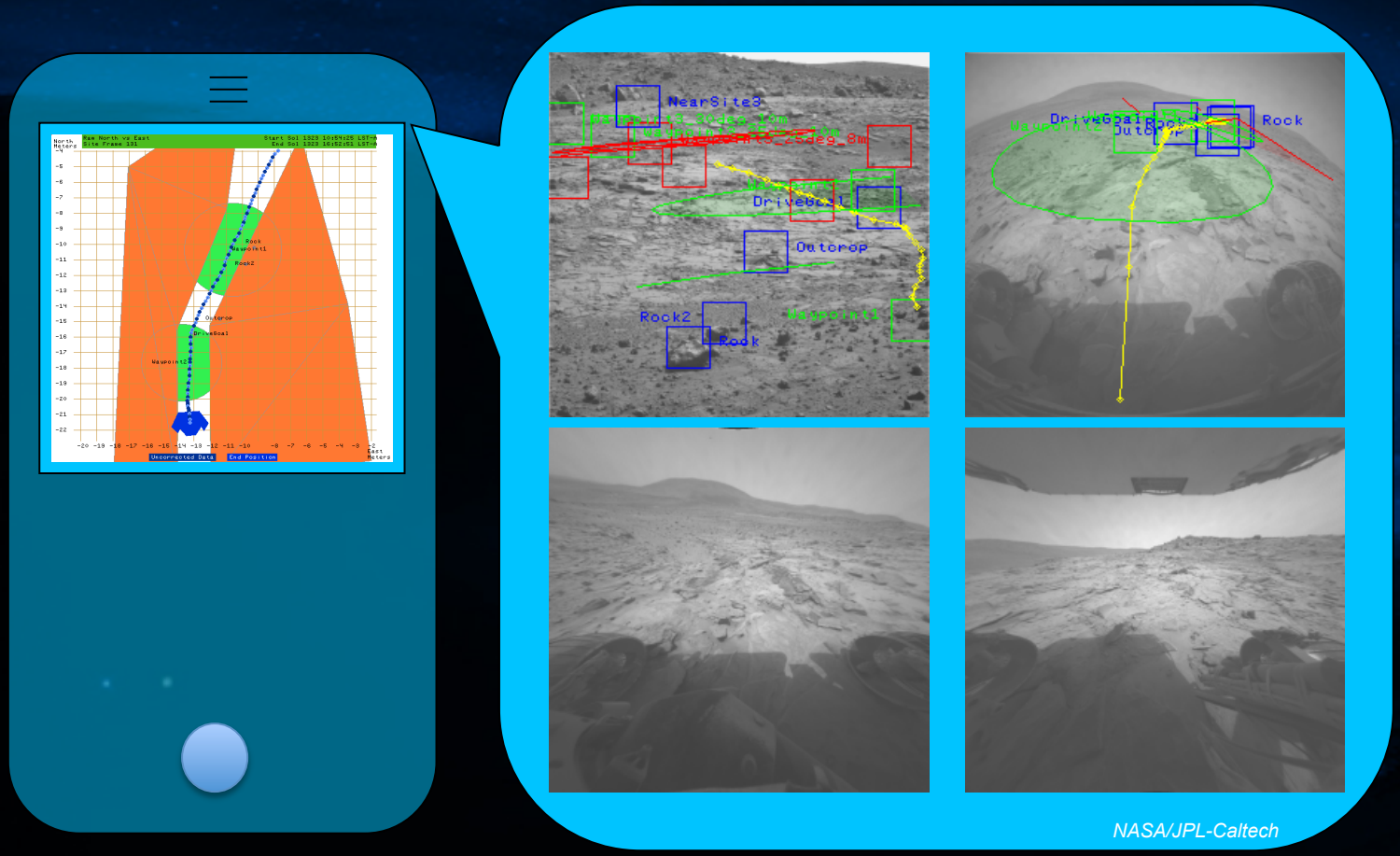
Curiosity Cumulative Odometry



NASA/JPL-Caltech


C++ is pervasive on Earth
NASA/JPL-Caltech
Much of the software used to plan Mars rover drives on Earth is also written in C++

C++ Annotates Drive Data



Automated graphical annotation of downlink data is done in C++, then automatically sent to the team's phones

C++ on Other Spacecraft

Earth Observing 1 – Autonomous Sciencecraft Experiment since 2005

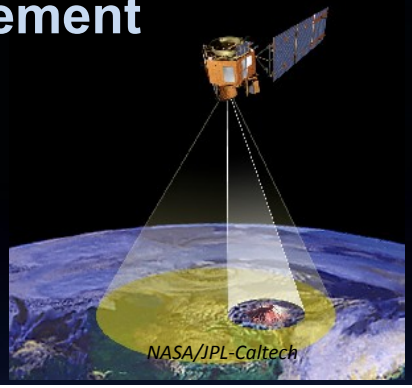
ISS-RapidScat – ocean wind velocity measurement

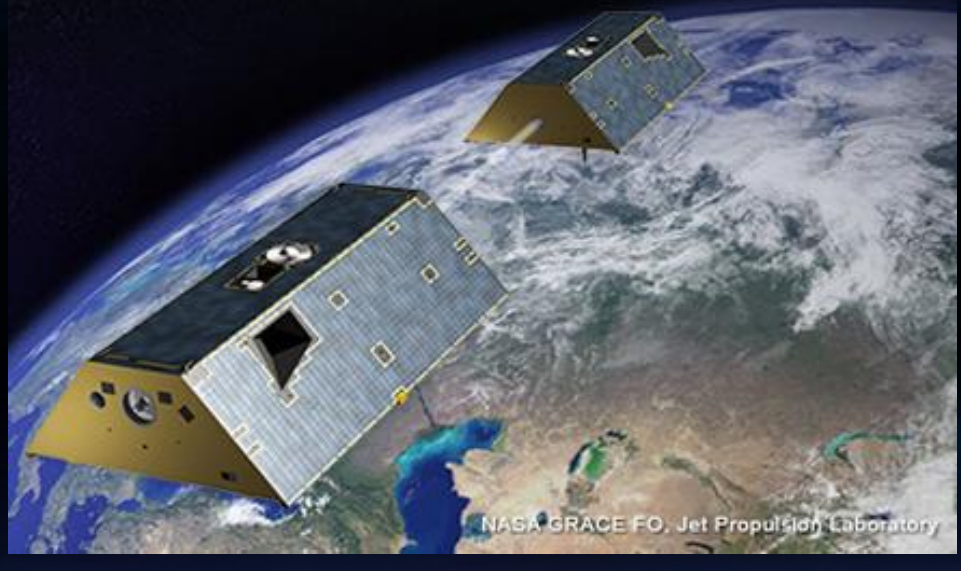
Aquarius – Sea Surface Salinity

Grace Follow-On – tracking water movement

Cubesats


NASA/JPL-Caltech/JSC


NASA/JPL-Caltech



The Future of C++ in Space

"With modern tools, C++ can be cheaper to validate than C"
– Rus Knight, Casper Cog E on EO1

MER and MSL paved the way for the 2020 Rover, which will inherit MSL's C++ code base

EO1 (Remote Agent, Aspen) and Aquarius already take advantage of more than just Embedded C++ constructs.

James Webb telescope is using C++ in the IBM Rational framework

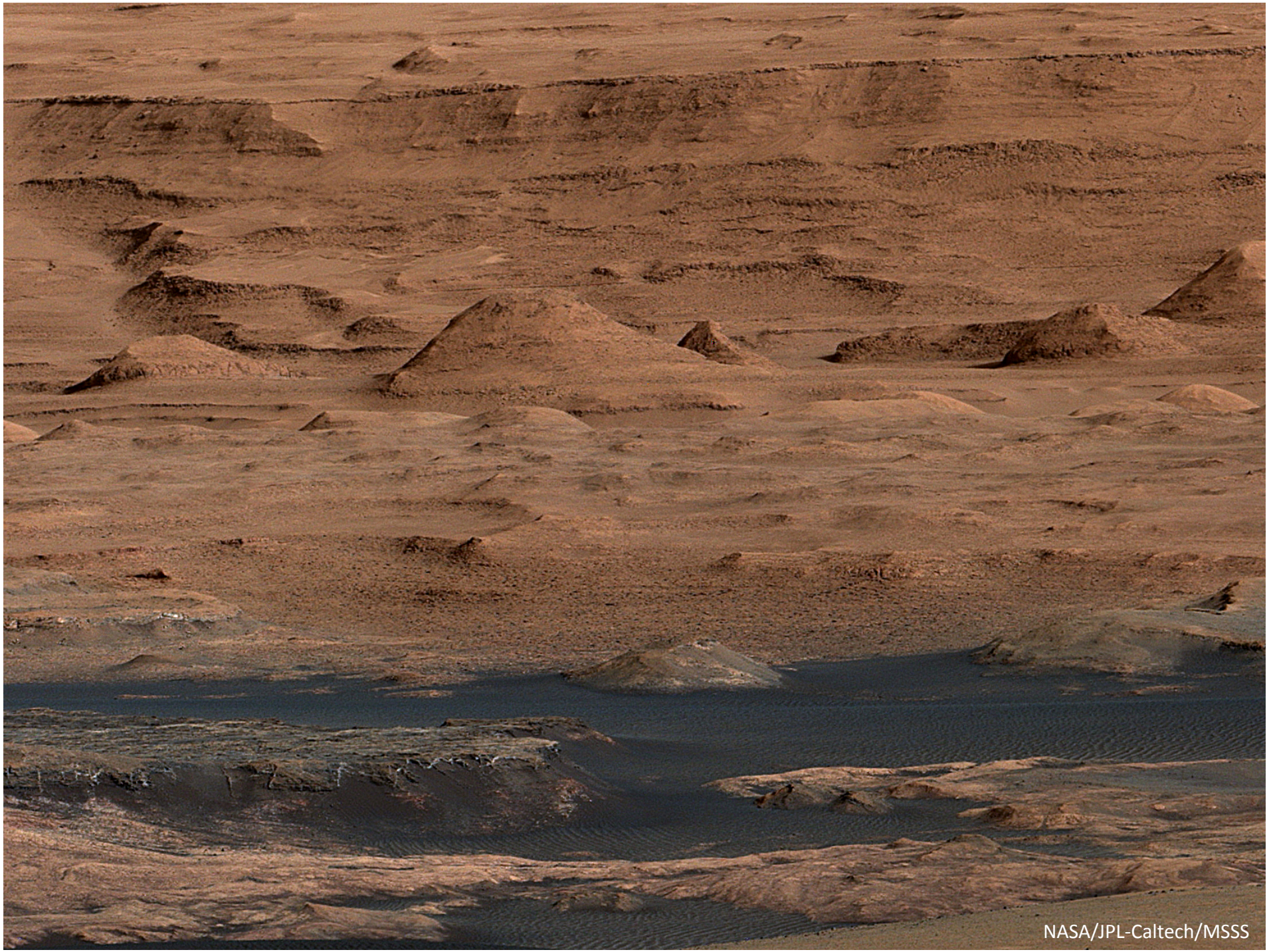
Grace FO and other projects are advocating for tighter integration with UML code parsers and generators

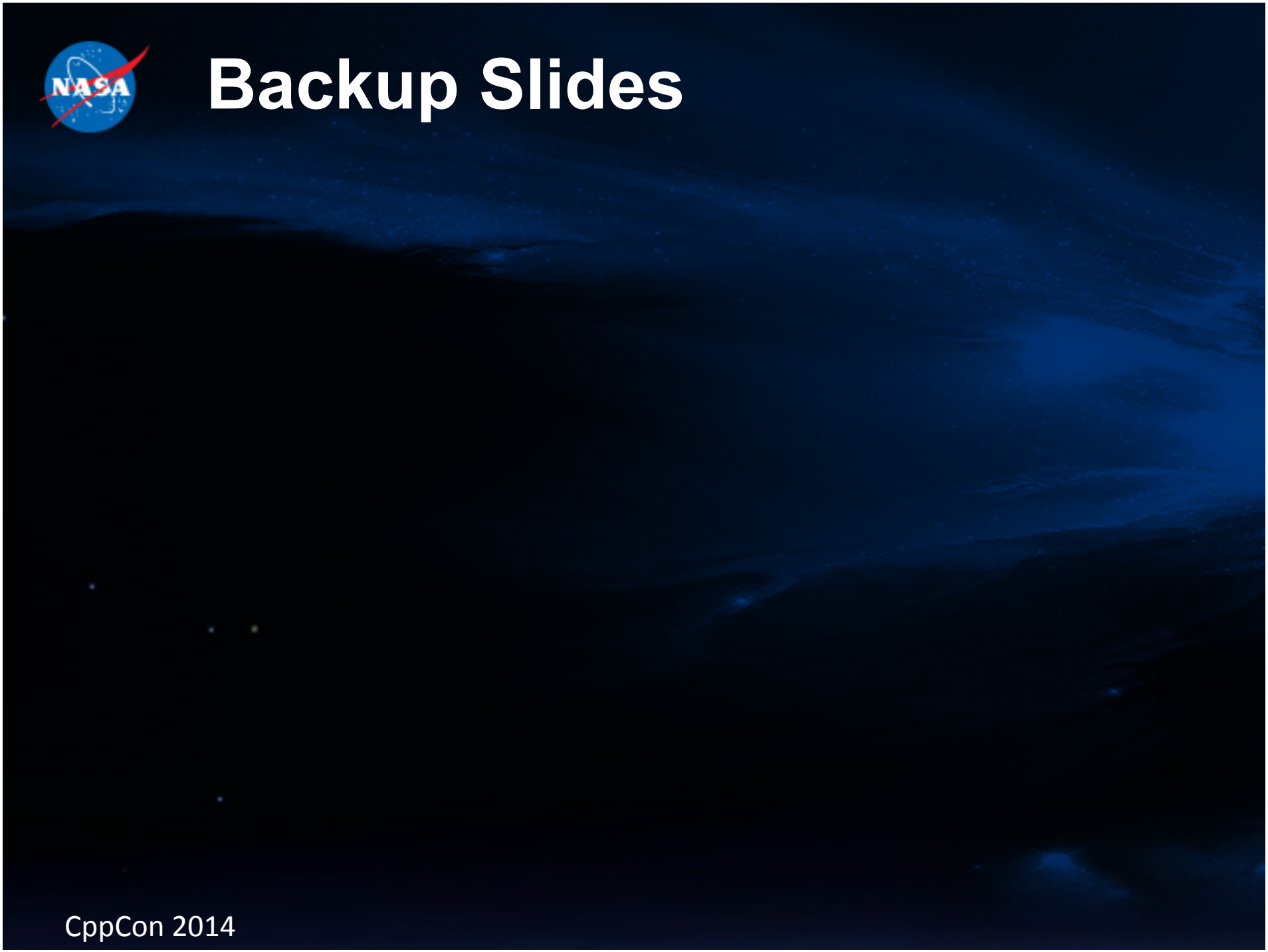
Learn more: *OO Techniques Applied to a Real-time, Embedded, Spaceborne Application*, Murray, Shahabuddin, OOPSLA 2006

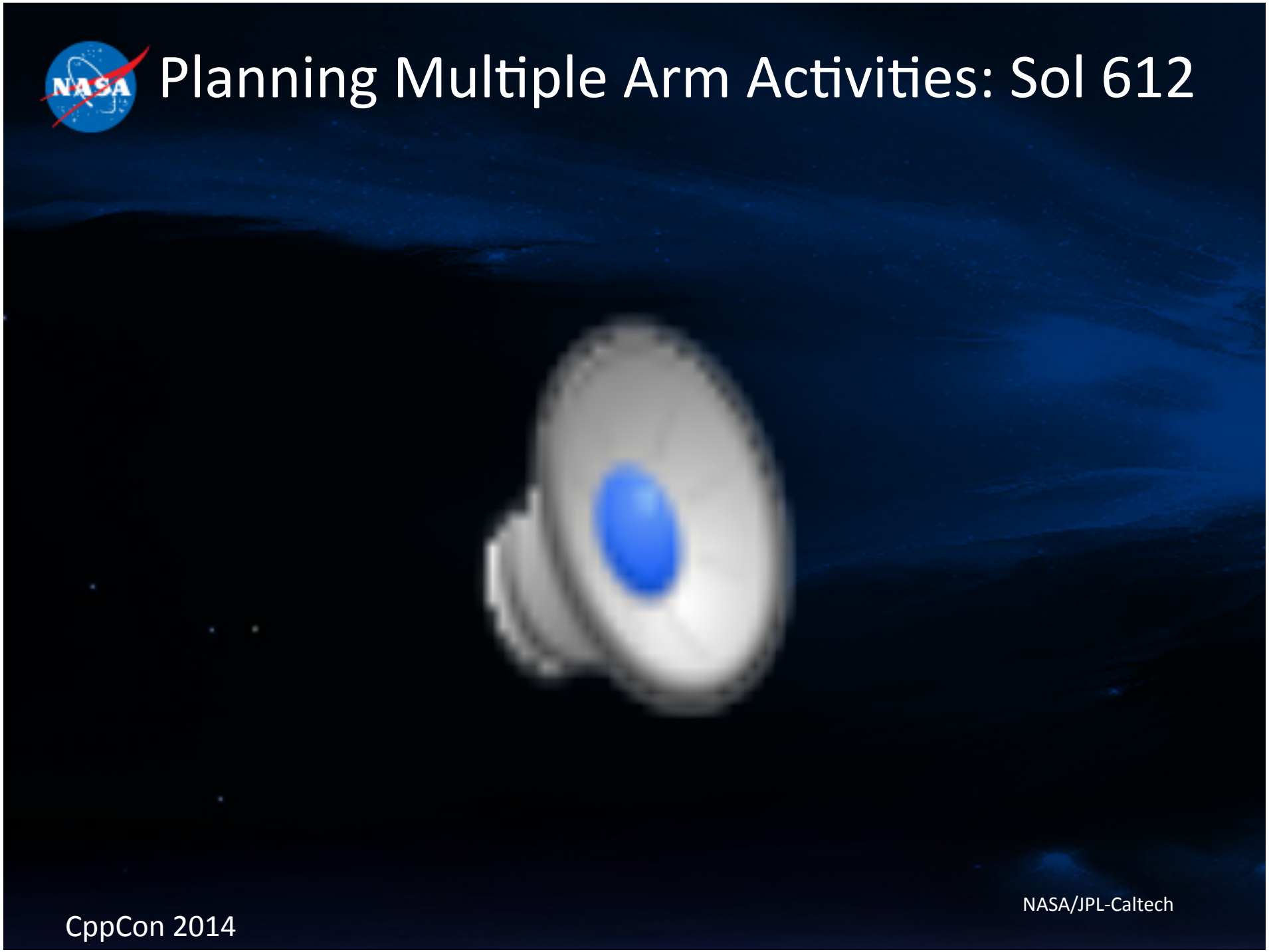
Acknowledgements

Thanks to multiple JPLers whose knowledge and opinions contributed greatly to this presentation, including but not limited to: Jeff Biesiadecki, Tim Canham, Dan Dvorak, Gerard Holzmann, Rus Knight, Mike McHenry, Issa Nesnas, Glenn Reeves, Steve Scandore, and especially Alex Murray (publish your white paper, Alex!).

The work described in this talk was performed at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the National Aeronautics and Space Administration (NASA).


NASA/JPL-Caltech/MSSS


Backup Slides
CppCon 2014


Planning Multiple Arm Activities: Sol 612
NASA/JPL-Caltech
CppCon 2014

Sol 122: VO vs IMU

- By convention, any VO updates that measure more attitude change than the IMU does will be rejected; we tend to trust the IMU, especially over short distances
- On Sols 122-124, Curiosity drove using Visual Odometry (VO), but several VO updates were rejected!
- Turned out that VO was right! A parameter caused the IMU gyro-based attitude estimator to reject changes under high accelerations
- No more issues since updating that parameter
- *VO updates have failed to converge just 13 times out of 3855 attempts as of sol 650, and only twice for actual lack of texture; 99.66% success rate!*