NASA’s Voyager is an exquisite object of exploration and of engineering.

From an explorer’s point of view, Voyager I is the first human artifact to have waded into the shores of interstellar space. Launched in 1977 and expected to keep operating through 2020, Voyager has witnessed active volcanoes on the moons of Jupiter, mapped heretofore-unknown regions of the heliosphere, and presented us with our most distant selfie, the image of the Pale Blue Dot, taken in 1990 from a distance of almost 40.5 AU, or slightly more than 6 billion kilometers from Earth.

From an engineer’s perspective, Voyager is a beautiful case study in the development of complex software-intensive systems. Voyager’s computers were an evolution of a proven system, the Viking Command Computer Subsystem (CCS). Because of the increased computational demands for its scientific mission, Voyager placed two instances of CCS on board—one for basic spacecraft hygiene and the other for attitude control—plus a third custom computer for flight data. As is typical for high-reliability real-time systems, each of these three systems was architected with two processors and built out of approximately 150 standard transistor-transistor logic integrated circuits. There existed a total of less than 70 Kbytes of memory for all three subsystems, and—in a first for space systems—the custom system used dynamic CMOS memory. Another important architectural decision—also novel for space systems—was the use of what at the time was called “soft logic.” Today, we call it firmware.

Voyager’s flight software was largely the product of just two men,

continued on p. 86
Richard Rice and Edgar Blizzard. As a software engineer, one thing that Rice said about his experience stuck out to me: “We didn’t worry about top-down or structured,” he said. “We just defined functions.” Indeed. Testable, measurable, executable software that works is always the most important artifact of any software-intensive project.

You can read more about the story of computing at NASA at http://history.nasa.gov/computers/contents.html.

**Design Serendipity**

Developing software-intensive systems is always an incremental and iterative process, both within the lifespan of a single system and across entire families of systems. Voyager built upon the experience of the Viking project (a planetary mission to Mars) as well as that of Pioneer (whose missions were to explore Jupiter, Saturn, and beyond). In retrospect, it’s clear that the impetus of both manned and unmanned space exploration led to pioneering discoveries in not just astrophysics but also in the best practices of systems engineering.

A number of serendipitous, unexpected connections punctuate the story of computing and space exploration. While at the Jet Propulsion Laboratory, Jim Blinn developed animations of the Voyager mission flybys, critical for reasoning about the spacecraft’s trajectory (http://youtu.be/SQk7AFE13CY). His work also served to visualize the project, helping to inspire an entire generation via Carl Sagan’s original Cosmos series. Among other things, Jim developed some of the fundamental algorithms in computer graphics, such as bump mapping and Blinn-Phong shading, and in so doing, he helped lay the foundation for modern computer graphics.

As for one of those curious connections, let’s look at the Phong shading algorithm and the Pioneer anomaly.

After the Pioneer 10 and 11 spacecraft completed their missions within the solar system, they continued on, but their trajectories weren’t what NASA expected. Rather, they were slowing down faster than predicted, and no one knew why. Eventually, with the help of a multitude of other researchers across the globe, Slava Turyshev reported that the heat of the spacecraft itself was causing the deceleration. “The effect is something like when you’re driving a car and the photons from your headlights are pushing you back,” he noted (www.jpl.nasa.gov/news/news.php?release=2012-209).

This “radiative momentum transfer” was modeled to a high degree of accuracy (http://arxiv.org/abs/1103.5222). As it turns out, the deceleration wasn’t just from heat directly radiating off the spacecraft and into space; it was also from heat radiating off the spacecraft, reflecting off its communication dish, bouncing back to the spacecraft, and then into space. How did the scientists figure this out? By applying the Phong shading algorithm.

As the Pioneer anomaly demonstrates, even the smallest forces, spread across time, can make a very big difference. Fortunately, software doesn’t weigh anything.

**Or does it?**

**In the Holes**

Barry Boehm tells the most wonderful story of the incredible lightness of software (www.sigsoft.org/SENS/boehm.html):

“*This is a story that I heard second-hand at TRW, but it may be an urban myth. It is about a weights engineer on a spacecraft. He accounts for all of the weight on a spacecraft. The weights engineer came to a software engineer and said, according to budget, there is $3,000,000 accounted for the software and I want to know how much it weighs. The software engineer replied, “Nothing.” The weights engineer said he wanted that kind of a job. You get paid well to produce something that weighs nothing. A week later, the weights engineer came back with a deck of cards. “Is there any software in here?” he asked. The software engineer said, “Yes.” The weights engineer said he would weigh the deck to determine the weight of the software. The software engineer said, “You have it all wrong; we only use the holes.”*”

Software: it’s in the holes. The trick, of course, is putting the holes in exactly the right place.

If you want to be very pragmatic, while software itself might not weigh anything, the memory that holds that software does. Therefore, the more software you pack on a spacecraft, the more it will (indirectly) weigh. Now, if you also want to be excruciatingly precise, the flipping of bits sheds thermal energy, and thus there will be very subtle differences in radiative momentum transfer when an onboard computer is computing and when it is not.

So, although software has no weight, you might have to design it to compensate for objects that do have weight and move. For example, imagine that your spacecraft is moving very, very fast—fast enough that you must account for Einstein’s special theory of relativity. This is the situation with GPS satellites: ignoring time dilatation, they would be off by 38 microseconds each day (www.astronomy.ohio-state.edu/~pogge/Ast162/Unit5/gps.html). As such,
Explaining Ourselves

From the inside of a software-intensive system across different domains, there are many different styles of implementation, each with its own subtle characteristics and idiosyncrasies. From the outside, however, especially to someone who isn’t a developer, this software in every domain appears not just weightless, but also invisible. To the outside, it’s all just holes.

How, then, does a computing insider explain to someone on the outside just how beautiful and elegant and dramatic and messy and wonderful it is on the inside? This is a conundrum. I suppose one must begin by attending to the more fundamental question of why anyone from the outside should care.

To attend to that second question first, I refactor Sagan’s classic statement, observing that we live in a society exquisitely dependent on computing, in which hardly anyone knows anything much about computing (www.cscि.org/si/show/why_we_need_to_understand_science). For our generation, and for those that follow, we all use the artifacts of computing, but now must learn the skills of computational thinking, a concept first described by Seymour Papert (www.papert.org/articles/AnExplorationinthetheSpaceofMathematicsEducations.html) and now further developed at Carnegie Mellon’s Center for Computational Thinking (www.cs.cmu.edu/~CompThink).

Back to the first question, I’d observe that one must start with some basic skills. It’s hard to unpack the enigma of Bach, the chaotic beauty of Jackson Pollock, or even the raw energy of Skrillex without having some underlying appreciation for their respective medium. So it is with software-intensive systems: you at least need to grok the language of the medium, and only then can you begin the journey of understanding the nuances.

What skills should we teach? Please understand that I’m out of my domain here: I’m not an educator, and I don’t have any credentials to formally know what works and what doesn’t. That being said, I’m encouraged by the reality that there’s an abundance of resources, from code.org’s Hour of Code (http://code.org/learn) to the Codeacademy (www.codeacademy.com) to even the Khan Academy (www.khanacademy.org/cs). For those with a more hands-on inclination, there’s the Raspberry Pi (www.raspberrypi.org) and the Arduino (http://arduino.cc). In all, we have an embarrassment of riches.

Yet, we must be realistic. Not every child needs to learn to code (but each one should learn some elements of computational thinking), and not everyone will care to become a programmer or software engineer. However, it’s a start, the start of a journey of understanding how something that weighs nothing can change the world. ☮

GRADY BOOCH is an IBM Fellow and one of the UML’s original authors. He’s currently developing Computing: The Human Experience, a major transmedia project for public broadcast. Contact him at grady@computingthehumanexperience.com.