The Wake Shield Facility (WSF) – a 12-ft-diameter free-flying platform – was deployed by the Space Shuttle to create an ultra-high vacuum wake in low Earth orbit [1]. Such experiments demonstrate the need for high-fidelity simulations that can model spaceborne physics (e.g. outgassing, orbital wake flows, surface deposition) with realistic detail and performance.

Modern High-Fidelity C++ Simulations for Spaceborne Systems

Space systems like WSF demand simulations that capture complex **spatiotemporal physics** (rarefied gas dynamics, thermal desorption, material deposition, etc.) while interfacing with modern machine learning (e.g. Graph Neural Networks). Below, we survey state-of-the-art **C++ physics engines**, discuss **graph-based/constraint-based simulation techniques**, outline **integration paths with ML pipelines**, and highlight **examples in aerospace & advanced manufacturing**. Each section provides key libraries, methods, and codebase examples, with references and code insights where relevant.

1. C++ Physics Simulation Engines (High-Fidelity & Performance)

Modern physics engines written in C++ provide the foundation for simulating rigid bodies, fluids, contacts, and multi-physics with high performance. Many are actively used in industry and research, often with GPU acceleration and Python bindings for flexibility. Key examples include:

- **Project Chrono** – An open-source C++ multi-physics engine known for accuracy and scalability [2]. Chrono supports rigid **multibody dynamics** (with joints/constraints), finite element flexible bodies, granular media, and fluid–solid interaction [3] [4]. It offers parallel computing (MPI) and GPU modules (e.g. Chrono::GPU for DEM) to handle large systems (e.g. millions of particles) efficiently. Chrono is **actively maintained** (latest v9.0) and has Python bindings (**PyChrono**) for easy scripting [5]. *For example, Chrono has been used to simulate planetary rovers driving on granular soil composed of millions of discrete particles, leveraging multi-core and GPU acceleration for real-time performance [4].*

- **NVIDIA PhysX** – A powerful, industry-proven physics SDK now fully open-source (BSD-3) [6]. PhysX provides highly optimized simulation of rigid bodies, collision, joints, and articulations, with a unified solver for multi-physics (rigid, soft bodies, fluids, cloth) [7] [8]. It runs on CPUs and GPUs, enabling scalable simulations from desktop to HPC clusters. PhysX's **GPU pipeline** (CUDA acceleration) achieves large speedups and is used in robotics and autonomous systems (e.g. NVIDIA Isaac Sim for digital twins [9]). Its **accuracy and stability** are well-established – supporting conserved momentum, robust stacking, and reduced-coordinate articulations (Featherstone solver) for complex jointed systems [10] [11]. *PhysX is widely adopted in aerospace visualization and training (e.g. NASA's Hybrid Reality labs) and in game engines, ensuring both high fidelity and real-time capability.*

- **SOFA (Simulation Open Framework Architecture)** – An open-source C++ framework focused on **interactive mechanical simulation** with multi-physics support [12]. SOFA emphasizes deformable models (using FEM) and real-time performance, making it popular in robotics and biomechanical simulations. It provides a plugin architecture to mix rigid bodies, soft tissues, and constraints, and a scene-graph system to easily construct simulation scenarios. Developed by an

active community (initially Inria, 2006), SOFA is efficient for prototyping **physics-based simulations** in research [13]. It also offers Python bindings (SofaPython3) and has been used for haptics and VR, which can translate to space applications (e.g. simulating robotic manipulators or flexible structures in microgravity).

- **Bullet Physics & MuJoCo** – Other notable C/C++ engines include **Bullet** (open-source, widely used in games and robotics) and **MuJoCo** (recently open-sourced by DeepMind). *Bullet* provides fast rigid-body and soft-body simulation with discrete and continuous collision detection. While primarily designed for games, it's used in robotics (PyBullet) and supports parallel solvers on CPU. *MuJoCo (Multi-Joint dynamics with Contact)* is a highly efficient engine tailored for model-based control and robotics; it's written in C (with a C++ wrapper) and known for accurate contact dynamics and speed. MuJoCo has been used in many reinforcement learning benchmarks and now being extended under open-source development [14]. Both Bullet and MuJoCo can be integrated into C++ projects or used via Python, and they emphasize stable simulation of jointed mechanisms – relevant for space robots, deployment mechanisms, or rover dynamics on other planets.

- **Project-specific Engines** – Some simulations in aerospace use specialized engines or frameworks: *NASA's Trick* (open-source simulation environment) provides a C++ **framework for time-based physics models**, handling common tasks like integration, real-time sync, and checkpointing [15]. Trick isn't a physics solver per se, but it integrates user-provided models with a robust architecture (for example, combining spacecraft dynamics, environmental models, etc.) [16]. Another is *ESA's DSpace or SIMSAT*, used for satellite operations, though those are less publicly accessible. In advanced manufacturing, commercial multiphysics tools (ANSYS, COMSOL) have C++ solver cores (often with Python interfaces) and can model thermal/vacuum processes, but open-source analogues (e.g. **OpenFOAM** for CFD) are also employed for custom simulations. *OpenFOAM*, while a continuum fluid solver, has been adapted for contamination modeling in spacecraft by customizing it for molecular flow regimes [17].

Why C++? These engines leverage C++ for performance (close-to-hardware control, multithreading, vectorization) and maintainability in large codebases. Many support **GPU offloading** via CUDA or OpenCL (PhysX, Chrono, some Bullet extensions), and distributed computing (Chrono with MPI, OpenFOAM MPI, etc.), which is crucial for simulating millions of particles or complex fluid flows in space systems. They also expose C APIs or Python wrappers, making them extensible and able to integrate with higher-level workflows (e.g. calling a C++ sim from a Python ML script). Overall, this ecosystem of C++ engines provides the building blocks to simulate spacecraft docking dynamics, orbital debris impacts, satellite thermal environments, or in the case of WSF – the interaction of a platform with the rarefied atmosphere – at high fidelity and speed.

2. Graph-Based & Constraint-Based Simulation Techniques

Modern simulations increasingly exploit **graph-based representations** and sophisticated constraint solvers to model complex physics. In traditional engines, physical systems are often represented as graphs of bodies (nodes) connected by joints or contacts (edges). Constraint-based solvers enforce physical laws (e.g. no penetration, joint limits) by solving equations, sometimes leveraging graph structure for efficiency. Two notable trends are: (a) advanced constraint solvers in physics engines, and (b) learned graph-based physics models from machine learning research.

- **Constraint Solvers in Engines:** Engines like Chrono and PhysX implement **Lagrange multiplier and iterative solvers** to satisfy joint and contact constraints. For example, Chrono can enforce

bilateral constraints (e.g. revolute joints) with high accuracy using complementarity solvers or iterative methods, ensuring stable multi-body simulations [18]. NVIDIA PhysX offers a *Reduced-Coordinate Articulations* model, which treats a kinematic chain as a tree graph and achieves linear-time solves with no joint drift [10]. It also provides **Position-Based Dynamics (PBD)** for systems like cloth, fluids, and granular matter [19] – PBD is a constraint-based method that iteratively adjusts positions of particles to satisfy constraints (distance, volume preservation, etc.), making it robust for complex, coupled phenomena. These methods draw on the graph structure of the system (e.g. a cloth mesh or a cluster of interacting particles) to propagate constraint corrections quickly. The result is **stable, realistic behavior** even in scenarios with many coupled parts – crucial for simulating things like a satellite's solar panel deployment with hinge constraints, or a cluster of regolith particles sticking under microgravity.

- **Graph Network Simulators (GNS):** Inspired by graph representations, **machine learning approaches** have emerged that learn physics directly from data. DeepMind's *Graph Network-based Simulators (GNS)* are a prime example: they treat each particle or object as a node in a graph and learn interaction forces via message-passing neural networks [20]. GNS has been shown to successfully emulate complex physics like fluid dynamics and granular flows, serving as a **surrogate simulator** that can be much faster than traditional solvers once trained [21]. For instance, a GNS model can learn to predict the trajectories of thousands of particles (liquid, sand, etc.) through time, given initial states, by mimicking a physics integrator with a learned function. These learned simulators have achieved accuracy on par with some numerical methods in research settings [22]. While most GNS implementations are in Python (using PyTorch or JAX), the concept influences C++ simulations too – one can integrate a trained GNS model into a C++ pipeline (see §3) or use graph-based data structures to organize simulation state for parallel processing. Another variant, **C-GNS (Constraint-based GNS)**, explicitly learns a *constraint function* (as a GNN) instead of direct next-state prediction [22]. At runtime it finds the state that satisfies this learned constraint, merging data-driven learning with a solver approach. This is analogous to how engines solve constraints: rather than predicting motion outright, they define conditions (e.g. non-penetration) and solve them. C-GNS demonstrates improved generalization and the ability to incorporate new physical constraints at test time [23]. Such techniques hint at future simulators that combine learned physics with traditional solvers – particularly useful for phenomena that are hard to model from first principles (like complex chemical outgassing reactions) but where data or experiments exist.

- **Graph-Based Representations in Custom Sims:** Outside of ML, users sometimes build custom simulators that explicitly use graphs to model relationships. For example, a thermal network model of a satellite can be viewed as a graph (nodes as components, edges as thermal couplings), solved by spreading heat in that network. Similarly, constraint graphs are used in multi-body dynamics: e.g. *Drake* (a robotics simulator by MIT) builds a graph of rigid bodies and joints, then uses sparse linear algebra exploiting that graph for simulation. These approaches aren't standalone libraries but design patterns – relevant when simulating something like an **orbital environmental network** (where nodes could be surface patches and edges transfer flux of molecules or heat). The flexibility of C++ allows representing such systems with graph data structures and using optimized libraries (Eigen, SuiteSparse, etc.) to solve the resulting equations efficiently.

- **Benefits and Use-Cases:** Graph and constraint-based methods ensure **physical consistency** (no violations of conservation laws or joint limits) and often are key to **real-time performance**. In a space context, consider simulating *orbital debris impact on a space station*: a graph-based breakage model could represent the station's structure as interconnected elements and break edges when stress exceeds a threshold, all while a constraint solver maintains connectivity until

failure. Another example is *attitude dynamics of multi-satellite constellations*, where each satellite's state is a node and formation-keeping links (virtual springs or control laws) are edges – a graph simulation could propagate interactions across the network efficiently. By leveraging graph structures, one can naturally integrate with GNN-based **learning algorithms** that observe the graph state and predict outcomes or optimal controls, blending first-principles simulation with AI (e.g. using a neural net to approximate an unknown aerodynamic disturbance force between satellites).

In summary, graph-based and constraint-based techniques are at the forefront of simulation technology. Classic engines use them under-the-hood for stability and speed, while new ML-driven simulators use graph neural networks to **learn physics models**. For developers of spaceborne system simulators, these approaches offer a way to model complex interactions (like **outgassing-induced molecular flows or chemical surface interactions**) in a structured way that is both computationally efficient and amenable to machine learning integration.

3. Integration with Machine Learning Pipelines (Python & GNN Integration)

To achieve the best of both worlds – high-performance C++ simulation and advanced ML (e.g. spatio-temporal GNNs) – modern workflows often **integrate C++ code with Python and machine learning frameworks**. Several pathways enable this integration:

- **Python Bindings for C++ Simulators:** Many C++ engines provide Python APIs, allowing the simulation to run within a Python process. Tools like **PyBind11** and **SWIG** make it straightforward to expose C++ classes to Python. For instance, *Project Chrono's PyChrono* wrapper lets users create simulation objects, step the physics, and retrieve data in Python [5]. The benefit is that Python's rich ecosystem (NumPy, Pandas, PyTorch, etc.) can be used for analysis or ML training loops on the fly [24]. As the Chrono docs note, one can "train AI neural networks with TensorFlow" while running Chrono, treating the physics engine as just another Python module [24]. Similarly, **SOFA** has *SofaPython3* for scripting, and NASA's *Basilisk* framework was designed with a Python interface controlling a C++ core [25] [26]. Basilisk's architecture actually uses Python to configure simulations and log data, but all heavy computations (orbital dynamics, sensor models, etc.) execute in C/C++ for speed [25] [26]. This approach (lightweight Python "glue" with a performant C++ backend) is powerful: you can embed a simulation in a training loop (e.g. reinforcement learning where each step calls the simulator, then an RL agent updates its policy in Python) or use Python ML libraries to optimize parameters until the simulation output matches mission data.

- **Embedding ML Models into C++ Simulations:** Conversely, one often needs to call an ML model *from* within a C++ simulation (for example, using a learned GNN to predict a force or a sensor reading at each timestep). Two prevalent solutions are **TorchScript** and **ONNX**:

- *TorchScript:* PyTorch offers TorchScript to serialize models into a form that C++ can load and execute using the *LibTorch* C++ API. A model (e.g. a Graph Neural Network trained to predict pressure distribution) can be **exported to a `.pt` file** and then loaded in C++ like: `torch::jit::Module model = torch::jit::load("model.pt");` followed by `model.forward(inputs)` to get predictions. TorchScript models run in a high-performance C++ environment (with GPU support if available) – effectively embedding the neural network into the simulation loop [27]. This is ideal for deploying learned components in a C++ codebase. For example, one could replace a costly chemical kinetics calculation with a neural net

approximation: the C++ sim at each step queries `model.predict(state)` to get, say, outgassing rates or aerodynamic coefficients, vastly speeding up computation while retaining accuracy learned from data.

- *ONNX Runtime:* The Open Neural Network Exchange (ONNX) format allows models from various frameworks (TensorFlow, PyTorch, etc.) to be saved in a standard format. **ONNX Runtime** is a C++ library that can load these models and run inference on CPU or GPU. Integrating ONNXRuntime into a C++ simulation means the sim can call `Ort::Run(...)` with the current state and get back ML-predicted outputs. ONNX Runtime is optimized and supports hardware accelerators [28], ensuring minimal overhead. The advantage of ONNX is framework-agnosticism: for instance, if one team develops a GNN in PyTorch and another in TensorFlow, both can be exported to ONNX and used interchangeably in the C++ simulation. Many aerospace companies use ONNX for deploying ML models into flight software and simulation because of this flexibility and the focus on **cross-platform performance** [29].

- **Co-Simulation and Data Pipelines:** In some cases, the simulation and ML training run as separate processes that communicate. For example, one might have a C++ **real-time simulator** (e.g. simulating a spacecraft attitude in a hardware-in-the-loop setup) and a Python process that receives telemetry and makes decisions (training or inference). Communication can be through sockets, RPC, or shared memory. While this is less tightly integrated, it allows using each tool in its native environment (e.g. a high-fidelity simulation on a dedicated server and a training script on another machine). Frameworks like **ROS 2** (Robot Operating System) or NVIDIA's **Isaac** provide middleware to connect simulations with AI modules. However, the trend with high-performance computing is to avoid the overhead of separate processes and instead use the aforementioned in-process methods (bindings or embedding) for faster throughput.

- **Example – Graph Neural Network in a Loop:** To illustrate, consider simulating *orbital debris collision* and using a GNN to predict fragment propagation. One could run a C++ dynamics simulation for the spacecraft and debris but at each timestep call a GNN (via TorchScript in C++) to estimate how debris pieces might further break apart or spread (learned from many hypervelocity impact experiments). The sim would then continue using that output as input for the next step. During training, one might run the whole thing under PyTorch's control: Python code generates scenarios, steps a bound C++ sim, compares results to ground truth, and backpropagates errors to update the GNN. This kind of **hybrid training** (learned model inside classical simulator) is increasingly feasible with the tools above.

- **Interfacing with Other ML Tools:** Apart from neural networks, integration might involve optimization libraries (Ceres, NLopt) or data handling (HDF5 for large simulation output, to feed into ML later). C++ sims can output data that Python ML code later ingests for training surrogate models (for instance, running thousands of vacuum plume simulations with a C++ DSMC code, saving the results, and then training a GNN to emulate plume dynamics). In the other direction, one can use ML to guide simulations – e.g. an RL agent controlling a simulated satellite's thrusters via Python, where each action is sent to the C++ sim, which returns the new state and a reward. This pattern is common in OpenAI Gym environments; indeed, tools like **SofaGym** wrap SOFA simulations in OpenAI Gym interfaces for reinforcement learning [30] [31].

In summary, robust integration techniques exist to join C++ simulations with machine learning: **Python bindings** make it easy to call C++ engines in training scripts, and formats like **TorchScript/ONNX** allow calling ML models within C++ sims for speed. This synergy is vital for modern space systems work – e.g. using neural nets to approximate hard-to-model physics (outgassing, plasma effects) inside high-fidelity orbital simulators, or using simulators to generate synthetic data to train AI for autonomy. The result is

a pipeline where simulation and ML **co-evolve**: simulations generate data for ML, and ML enhances the simulation's capabilities.

4. Applications in Aerospace & Advanced Manufacturing

Bringing these elements together, we see cutting-edge simulations being applied to aerospace problems – from spacecraft environmental effects to in-space manufacturing – using modern C++ engines and often coupling with ML. Some **notable projects and examples**:

- **Spacecraft Contamination & Vacuum Physics:** The Wake Shield Facility's mission – producing ultra-high vacuum (UHV) for semiconductor-quality thin films – exemplified challenges like material outgassing and molecular deposition in orbit. Simulation of such phenomena requires rarefied gas dynamics tools. One approach is **Direct Simulation Monte Carlo (DSMC)**, a particle-based method for free-molecular flows. **SPARTA** is an open-source C++ DSMC code designed for low-density gas simulation on parallel systems [32] [33]. It can model outgassing by emitting test particles from surfaces and computing their trajectories and collisions in a vacuum chamber or open space. SPARTA (developed at Sandia) supports 3D triangulated surfaces (e.g. satellite geometry) and runs efficiently on MPI and even GPUs (via Kokkos), making it suitable for simulating something like the *orbital wake behind WSF* with millions of simulated molecules. In fact, studies in the 1990s used DSMC to predict "return flux" contamination on WSF – molecules outgassed from the spacecraft that scatter and redeposit on its surface [34]. Today, a tool like SPARTA could replicate those results with far greater compute power, or one could train a GNN surrogate from a set of DSMC runs to instantly predict contamination levels for various conditions.

- **Molflow+ (Monte Carlo Vacuum Simulation):** An exciting development in UHV modeling is CERN's **Molflow+** software. Written in C++ with an interactive GUI, Molflow is tailored for molecular flow (UHV) simulations and has become *"the de-facto industry standard for ultra-high-vacuum simulations."* [35] It uses a **test-particle Monte Carlo** approach: billions of molecule trajectories are traced through a geometry to compute pressure and flux distributions [36]. Molflow was originally developed for particle accelerators, but it has been applied to aerospace: for example, the German satellite manufacturer OHB adapted Molflow to simulate molecular contamination transport on satellites [37] [38]. They extended it with chemical processes (surface sticking and desorption) and cluster computing support [39]. Notably, Molflow was used in conjunction with NASA JPL codes to estimate return flux in planned Europa orbiter fly-bys [38]. Molflow is now open-source (GPL) and can be scripted or extended, making it a handy tool to integrate with ML (one could generate many random simulations to train a model, or use ML to optimize vacuum chamber designs via Molflow in the loop). For space manufacturing, where **thin film deposition** or 3D printing in vacuum is considered, such Monte Carlo simulators can predict how vapor spreads and sticks to surfaces. Molflow's success shows the value of specialized high-fidelity simulators – and by being open and scriptable, it fits well into modern pipelines (e.g. one could wrap Molflow in Python to perform design-space exploration with Bayesian optimization).

- **Satellite Dynamics and Mission Simulation:** Simulation frameworks like **Basilisk** (open-source, C++/Python) are used to model whole-spacecraft dynamics and environments [25] [40]. Basilisk can simulate orbital motion, attitude control, sensor readings, and even multi-satellite constellations, all with high speed (it aims for faster-than-real-time, e.g. simulating a year in a day) [26]. It achieves this by C++ optimization and parallelization while the user interacts via Python. Basilisk has been applied to real missions and supports hardware-in-the-loop, so it's a

good example of an actively maintained codebase in aerospace. While Basilisk's focus is on rigid-body dynamics and control systems, it can be extended to include environmental physics (one could integrate a sub-module for plume impingement or thermal analysis). The **modular design** (each aspect of the spacecraft is a module with a clear interface) makes it feasible to plug in learned components – e.g., a neural network that estimates orbital drag based on space weather could replace a simple analytic model. The framework's messaging system (data bus) and Python analysis capabilities [41] mean it can easily log large amounts of simulation data for training ML models (for autonomous planning, fault detection, etc.). Basilisk exemplifies how **modern C++ simulation frameworks** are built with integration in mind: it uses SWIG to expose C++ to Python, supports cluster runs for Monte Carlo, and has hooks for real-time operation, covering the needs of both AI development and classical engineering analysis in one environment.

- **NASA's Contamination Analysis Tools:** Historically, NASA developed programs like **NASAN** (a molecular contamination analysis code for ISS) which could model deposition from thruster plumes and material outgassing [42] [43]. NASAN (written in Fortran/C) used ray-tracing to account for geometry shadowing and view factors, and could handle 25,000 surface elements – achieving higher fidelity than previous tools [44]. While dated (circa 2000), it demonstrated modular design (plume models, column density calculators, etc. in one package) and even some parallel processing for ray-tracing [45] [46]. Modern equivalents would leverage C++ multithreading/GPU for ray-tracing; in fact, some teams have integrated **game engine ray-casting** (like using NVIDIA OptiX) to accelerate line-of-sight computations in space sims. The key takeaway is that high-fidelity contamination modeling is a multi-physics problem – vacuum kinetics, surface chemistry, spacecraft geometry – and thus benefits from **graph-based approaches** (each surface or source as a node in a network exchanging molecules) and ML (to approximate complex physics). For example, one could train a model on NASAN outputs to quickly predict deposition on a new geometry, saving design iteration time.

- **Advanced Manufacturing in Space:** Simulating manufacturing processes (like welding, sintering, or crystal growth in microgravity) can involve coupling thermal, fluid, and structural models. C++ multiphysics libraries (e.g. **SALOME or OpenFOAM** for CFD, **Deal.II** for FEM, etc.) can be combined to tackle these. A concrete example: simulating a **laser metal deposition** 3D printing process in orbit might use an OpenFOAM-based solver for the carrier gas flow (if any) and a custom particle simulation for metal powder flow, plus a thermal FEM for the melt pool. Each of these might be separate C++ solvers coupled via scripts or a common interface. This is where frameworks like **Project Chrono** or **SOFA** can help: Chrono, for instance, has been used in granular material simulations relevant to manufacturing (e.g. powder mixing [47]) and offers co-simulation interfaces to couple with CFD packages [4]. By running these in parallel (and potentially training a **Graph Neural Net** to act as a coordinator or surrogate), one could achieve near-real-time predictive simulations of the manufacturing process. On the ML side, **spatio-temporal GNNs (ST-GNNs)** are well-suited to model the evolving state of such systems – e.g. nodes representing different regions of a build plate with edges for heat transfer and material flow. Integrating an ST-GNN that has learned from high-fidelity simulation data could enable fast forecasting of defect formation or material properties in-situ, which is invaluable for control systems that must adjust parameters on the fly.

- **Case Study – AI-Enhanced Thermal Control:** Consider a satellite with a complex heat pipe network. A C++ thermal simulation (using, say, a finite-volume method) can capture the physics, but model reduction via ML can speed up design trades. One approach demonstrated in research is training a GraphNet to emulate thermal simulations across configurations, allowing near-instant evaluation of different designs [48] [49]. This mirrors DeepMind's MeshGraphNets concept, which has shown success on fluid and structural mechanics problems. In practice, one

could use a C++ solver to generate many samples (varying heat loads, etc.), train a GNN in Python on that data, then deploy the GNN in a C++ thermal analysis tool for real-time estimation or control. This synergy exemplifies the future of aerospace simulations – a tight loop between high-fidelity C++ models and learning algorithms that generalize those results.

In conclusion, building a high-fidelity simulator for space systems like WSF today involves **choosing a robust C++ physics engine**, incorporating advanced **graph-based modeling or learned components** for the specialized physics (molecular flows, etc.), and ensuring **seamless integration with ML pipelines**. Fortunately, the tools and libraries available – Chrono, PhysX, SOFA for the core mechanics; GNN-based simulators from DeepMind's research; PyBind11, TorchScript, ONNX for ML integration; and domain-specific codes like SPARTA or Molflow for vacuum physics – provide a rich ecosystem to draw from. Aerospace projects are already leveraging these: from **UHV contamination analysis with Monte Carlo codes enhanced by open-source contributions** [38] [39], to **spacecraft mission simulators coupling C++ speed with Python AI libraries** [25] [40]. By combining rigorous physical modeling (e.g. DSMC for outgassing) with modern computational techniques (GPU acceleration, distributed computing) and augmenting with machine learning (GNN surrogates, control policies), one can achieve simulations that are both **highly realistic and computationally tractable**. This empowers engineers to test scenarios that were previously infeasible and to integrate simulation tightly with design optimization and autonomous control development – a key step as we venture into more complex spaceborne systems and in-orbit manufacturing in the years ahead.

Sources: Supporting references are provided inline, citing documentation, research, and examples that substantiate the described methods and tools. Each citation (e.g.【6】,【18】,【40】) corresponds to an external source with additional details.

---

1 File:Wake shield facility.jpg - Wikimedia Commons
https://commons.wikimedia.org/wiki/File:Wake_shield_facility.jpg

2 3 4 5 18 24 47 Project Chrono - An Open-Source Physics Engine
https://projectchrono.org/

6 7 8 9 10 11 19 PhysX SDK - Latest Features & Libraries | NVIDIA Developer
https://developer.nvidia.com/physx-sdk

12 13 SOFA
https://www.sofa-framework.org/

14 MuJoCo — Advanced Physics Simulation
https://mujoco.org/

15 16 Trick | ATutIntroduction.md
https://nasa.github.io/trick/tutorial/ATutIntroduction.html

17 Contamination modeling using the OpenFOAM open source library
https://medicalimaging.spiedigitallibrary.org/proceedings/Download?urlId=10.1117%2F12.2322156

20 21 GitHub - geoelements/gns: Graph Network Simulator
https://github.com/geoelements/gns

22 23 [2112.09161] Constraint-based graph network simulator
https://arxiv.org/abs/2112.09161

25 26 40 41 Welcome to Basilisk: an Astrodynamics Simulation Framework — Basilisk 2.7.17 documentation
https://avslab.github.io/basilisk/

27 TorchScript for Deployment — PyTorch Tutorials 1.8.1+cu102 documentation
https://h-huang.github.io/tutorials/recipes/torchscript_inference.html

28 mohamedsamirx/YOLOv12-ONNX-CPP - GitHub
https://github.com/mohamedsamirx/YOLOv12-ONNX-CPP

29 C++ | onnxruntime
https://onnxruntime.ai/docs/get-started/with-cpp.html

30 An Open Platform for Reinforcement Learning Based on Soft Robot ...
https://pubmed.ncbi.nlm.nih.gov/36476150/

31 SofaGym: An Open Platform for Reinforcement Learning Based on ...
https://www.liebertpub.com/doi/abs/10.1089/soro.2021.0123

32 33 SPARTA Direct Simulation Monte Carlo Simulator
https://sparta.github.io/

34 Ronald M. Sega's research works | Johnson Space Center and other ...
https://www.researchgate.net/scientific-contributions/Ronald-M-Sega-6090834

35 36 37 38 39 48 49 Tracing molecules at the vacuum frontier – CERN Courier
https://cerncourier.com/a/tracing-molecules-at-the-vacuum-frontier/

42 43 44 45 46 Software for Analysis of Contamination of the ISS - Tech Briefs
https://www.techbriefs.com/component/content/article/6916-msc-22963