ChatGPT

Newtonian Mechanics

- **Libraries/Frameworks:** Use physics engines like **Bullet** (open-source C++ collision and rigid-body simulator [1]) or **ODE/PhysX** for rigid-body and collision handling. For math, use **Eigen** or **GLM** for vectors and matrices. For GPU acceleration, **CUDA** or **OpenCL** can parallelize particle updates. For visualization, use **OpenGL** (via **GLFW** or **GLUT**) or multimedia libraries like **SFML** to render particles/bodies. For example, one CUDA-based demo uses GLFW+GLM+SFML with CUDA to simulate 10,000 colliding spheres at 144 FPS on an RTX 3080 [2] [3].
- **Numerical Methods:** Integrate Newton's equations with time-stepping methods. Simple explicit Euler is easy but unstable, whereas *symplectic* integrators (e.g. Verlet) preserve energy better [4]. Higher-order methods like 4th-order **Runge–Kutta** (RK4) give more accuracy per step (used in many engines [5]). For example, a physics project explicitly "use[s] Runge-Kutta 4th Order to integrate bodies" [5]. Broad-phase collision detection (e.g. spatial hashing or sweep-and-prune) pairs with narrow-phase impulse solvers or constraint solvers to handle collisions. Rigid bodies require solving translational and rotational dynamics (use inertial tensors and impulse resolution). Note the trade-off: simple Euler is fast but noisy, while RK4 or Verlet are slower per step but more stable and accurate for long simulations.
- **Example (C++ pseudo-code):** A simple Verlet/explicit Euler update for particles might look like:

```
Vec3 accel = computeForces(pos, vel) / mass;          // e.g. gravity,
springs
vel += dt * accel;                                    // semi-implicit
Euler
pos += dt * vel;                                     // update position
```

  For improved accuracy, one could implement RK4:

```
VecState y = {pos, vel};
auto f = [&](double t, const VecState &s){ return
computeDerivatives(s); };
VecState k1 = f(t, y);
VecState k2 = f(t+dt/2, y + k1*(dt/2));
VecState k3 = f(t+dt/2, y + k2*(dt/2));
VecState k4 = f(t+dt,   y + k3*dt);
y += (k1 + 2*k2 + 2*k3 + k4)*(dt/6);
pos = y.pos; vel = y.vel;
```

  (Libraries like **Eigen** can help build the matrices/vectors for such integrators.)
- **Visualization:** Render particles and rigid bodies using graphics APIs. For 2D, you might draw circles with OpenGL or **SDL2**; for 3D, draw spheres or meshes. Use **ImGui** overlays to tweak parameters (mass, restitution, etc.) in real-time. GPU compute shaders or CUDA kernels can update positions of thousands of particles in parallel, then map the results to vertex buffers for OpenGL rendering. Color or trail effects can emphasize velocity or energy.

Electromagnetism

- **Libraries/Frameworks:** Numerical E&M often uses grid-based or spectral libraries. Libraries like **Eigen** or **FFTW** assist with linear algebra and FFT. For Maxwell's equations, open-source C++ tools include **Meep** (a C/C++ FDTD solver with Python interface) [6] and **OpenEMS** (MPI-parallel C++ FEM solver) [7]. GPU computing (CUDA/OpenCL) can accelerate grid updates or particle-in-cell (PIC) methods. For visualization, use OpenGL or Vulkan to draw field lines or color-mapped field magnitudes; **ImGui** can overlay vector field arrows or sliders for sources.
- **Numerical Methods:** A common approach is the **Finite-Difference Time-Domain (FDTD)** method (Yee grid): discretize space-time and update E and B fields in a leapfrog manner [8]. This explicit scheme alternates solving Maxwell's curl equations (updating E then B), subject to the CFL stability condition. FDTD is easy to parallelize and extends naturally to GPU or SIMD [9]. For complex geometries, **Finite-Element** or **Boundary-Element** methods (FEM/BEM) solve the variational form, often with sparse linear solvers. For static fields (electrostatics), one might solve Poisson's equation via **Finite Differences** or spectral methods. Note the trade-offs: FDTD has simple updates and works well for broadband pulses [10], but can require small time steps and uniform grids, leading to large compute grids for high frequencies.
- **Example (Yee FDTD snippet):** A 1D FDTD update of fields might look like:

  ```
  for (int i = 1; i < N-1; ++i) {
    E[i] += (dt/eps) * (Hz[i] - Hz[i-1]) / dx;
  }
  for (int i = 0; i < N-1; ++i) {
    Hz[i] += (dt/mu) * (E[i+1] - E[i]) / dx;
  }
  ```

  (For 2D or 3D, you'd update each vector component similarly.) Libraries like **Eigen** can help assemble the field arrays, and **CUDA** can parallelize these updates across grid points.
- **Tutorials/References:** For more on implementing Maxwell solvers, see Georgii Evtushenko's post, which uses a compile-time C++ FDTD simulator [11]. The EpsilonForge blog surveys FDTD vs FEM trade-offs [8] and lists open-source packages (Meep, OpenEMS, gprMax) [6].
- **Visualization:** Visualize fields as textures or vector plots. For example, render the electric field magnitude on a 2D grid by mapping `|E|` to color. Use GLSL shaders to animate wave propagation in realtime. For magnetic fields or force lines, draw vector arrows or streamlines. Interactive controls can adjust charges/currents and show the resulting fields. ImGui plots can show field values along a line or energy density over time.

General Relativity

- **Libraries/Frameworks:** GR is most naturally done with general-purpose math libraries. Use **Eigen** or **GLM** (with double precision) for tensors and vectors. There's no widely-used simple GR "engine," so one typically implements geodesic integrators from scratch. **Boost::odeint** or custom RK solvers handle ODE integration. For visualization, graphics frameworks like **OpenGL/GLFW** (as in black hole demos [12]) or compute shaders can render lensing effects. For example, a recent C++17 project uses GLM for math and GLSL shaders to simulate photon geodesics in a Schwarzschild metric [12] [13].
- **Numerical Methods:** Simulate spacetime curvature by solving geodesic equations: $$\frac{d^2 x^\mu}{d\tau^2} + \Gamma^\mu_{\alpha\beta} \frac{dx^\alpha}{d\tau} \frac{dx^\beta}{d\tau} = 0.$$

Here $\Gamma^\mu_{\alpha\beta}$ are Christoffel symbols of the metric (e.g. Schwarzschild or Kerr). Numerically integrate these coupled ODEs with RK4 or adaptive integrators. Indeed, RK4 is reported to give "high stability and accuracy by averaging four derivative estimates per step" in such simulations [14]. One can integrate many light rays (null geodesics) in parallel: for instance, an OpenGL-based black hole renderer solves each ray's geodesic with RK4 [15]. The main trade-off is step size: smaller steps are needed near strong curvature to avoid instability or energy drift [14].

- **Example (Pseudo-code):** Integrating a Schwarzschild photon path:

```
Vec4 gamma[4][4]; // precompute Christoffel symbols for Schwarzschild metric
Vec4 x, u;        // position and 4-velocity (dx^μ/dλ)
double dλ = 0.01;
for (int i = 0; i < steps; ++i) {
  Vec4 accel = -contract(gamma, u, u); // compute -Γ^μ_{αβ} u^α u^β
  // RK4 steps for x and u (omitted for brevity)
  // e.g.: u_next = u + dλ * accel; x_next = x + dλ * u;
  // ...
}
```

Many implementations GPU-accelerate this by dispatching each ray's integration to a shader or CUDA thread [15].

- **Visualization:** Ray-traced rendering is common: cast rays through a distorted view, integrate geodesics, and color by escape direction or intensity. In practice, use **compute shaders** for per-pixel geodesic integration [15], then display the result as a full-screen textured image. Simple 2D modes can plot orbits in the equatorial plane. Use OpenGL or Vulkan to draw Einstein rings, light bending, and black hole silhouettes. For UI, ImGui can overlay controls for mass or camera position. A recent C++ OpenGL demo produced both 2D and 3D visualizations of gravitational lensing, leveraging shaders for speed [15] [16].

Quantum Mechanics

The time evolution of a quantum wavefunction (e.g. an electron in an infinite potential well) follows the Schrödinger equation. A pseudo-spectral **split-step Fourier** method is popular for solving the time-dependent Schrödinger equation [17]: alternate applying the potential's phase factor in real space and the kinetic phase in momentum space via FFTs. For example, the split-step (Fourier) method treats linear and nonlinear parts separately [17]. C++ libraries like **QEngine** (C++) provide 1D Schrödinger solvers for single/two-particle systems [18], and **QuEST** is a GPU-accelerated library for state-vector evolution [19].
- *Numerical methods:* Besides split-step, **finite-difference** schemes (e.g. Crank–Nicolson) can integrate the wavefunction stably. Crank–Nicolson leads to a tridiagonal linear system $A\psi(t+\Delta t) = B\psi(t)$ at each step [20], solvable in $O(N)$ time via the Thomas algorithm [20]. Spectral (FFT) methods require global operations but are very accurate for smooth wavefunctions, while finite-difference is local and easier to implement. Time–energy uncertainty means time steps and grid spacing must resolve the dynamics (smaller Δt for faster oscillations).
- *Example code:* Using split-step in 1D:

```
// psi: complex wavefunction array, V: potential array, k2: k^2 array for FFTs
```

```
for(int t=0; t<steps; ++t){
    // apply potential (real space)
    for(int i=0; i<N; ++i){
        psi[i] *= std::exp(-Complex(0, V[i])*dt/2);
    }
    // FFT to momentum space
    fft(psi);
    // apply kinetic (momentum space)
    for(int i=0; i<N; ++i){
        psi[i] *= std::exp(-Complex(0, k2[i])*dt/2);
    }
    // inverse FFT back to real space
    ifft(psi);
    // apply potential again
    for(int i=0; i<N; ++i){
        psi[i] *= std::exp(-Complex(0, V[i])*dt/2);
    }
}
```

(Alternatively, one can assemble A and use Eigen's sparse solver for Crank–Nicolson.)
- *Visualization:* Display probability density $|\psi(x,t)|^2$ as a surface or heatmap, evolving over time. For example, draw $|\psi|^2$ vs. x in real-time using OpenGL (points or a smooth curve). To illustrate interference, simulate a double-slit and project $|\psi|^2$ on a screen. GPU shaders can animate the wavefunction on a texture. Use ImGui or SDL to plot 1D slices or histograms of $|\psi|^2$. In the example above, the animation of an infinite well shows the wave bouncing inside the box (see figure).

Thermodynamics & Statistical Mechanics

- **Libraries/Frameworks:** Use general-purpose C++ libs (Eigen, Boost) for maths and random numbers. Monte Carlo simulations can use **Boost.Random** or **cuRAND** on GPU. For fluid simulations, consider **Lattice Boltzmann** libraries (e.g. Palabos or OpenLB). The **QuEST** framework above can also handle many-body quantum stats (Bose–Einstein condensates via Gross–Pitaevskii) [18] . For parallelism, **OpenMP** and CUDA help simulate many particles.
- **Numerical Methods:** A classic approach is **molecular dynamics**: simulate many particles (ideal gas molecules) with elastic collisions. Integrate Newton's laws (e.g. Verlet) and handle collisions via momentum exchange. Alternatively, **Direct Simulation Monte Carlo (DSMC)** randomly selects collision partners to approximate gas behavior. To sample equilibrium distributions, use **Metropolis–Hastings Monte Carlo**: generate microstates with probability ∝ e^(−E/(kT)). In fact, the Metropolis algorithm is a Markov-chain Monte Carlo method for sampling from such distributions [21] . For fluids, **Lattice Boltzmann methods (LBM)** solve discretized kinetic equations on a grid; modern C++ LBM code achieves GPU-level performance equivalent to thousands of CPU cores [22] .
- **Example:** To demonstrate Maxwell–Boltzmann statistics, one can sample particle speeds with a normal distribution in each velocity component. In code:

```
std::mt19937 rng(seed);
double kT = 1.0; // temperature
std::normal_distribution<double> dist(0.0, sqrt(kT));
vector<Particle> gas(N);
```

```
for(int i=0; i<N; ++i){
    gas[i].vx = dist(rng);
    gas[i].vy = dist(rng);
    gas[i].vz = dist(rng);
}
```

The resulting speed distribution follows the Maxwell–Boltzmann law [23]. As the simulation evolves (particles bouncing in a box), measure velocities and plot a histogram to verify this.

- **Visualization:** Render gas particles as moving dots or spheres in a container; color them by kinetic energy or speed. Display dynamic plots: e.g., a histogram of speeds versus the theoretical Maxwell–Boltzmann curve [23]. For energy/entropy, plot macroscopic quantities (pressure, temperature) over time. For fluids or LBM, render density/velocity fields as color maps. Use OpenGL point sprites for many particles, or map particle positions to a texture for 2D density plots. ImGui charts or SDL can overlay real-time graphs of energy distributions or pressure vs. time.

Trade-offs: Across all domains, note that GPU/parallel methods greatly speed up large-scale simulations (see [2] [22]). However, they often require simpler, local algorithms (e.g. explicit time-stepping) and careful handling of memory transfers. In contrast, more accurate implicit or global methods (like solving large sparse systems) may be slower but handle stiffness and large time steps. Choosing methods depends on desired accuracy vs. performance.

Sources: The above approaches are informed by physics and computational literature (e.g. Verlet integration's stability [4], RK4 use in physics engines [5], FDTD and LBM overviews [8] [22], etc.) and example C++ implementations [24] [13].

---

[1] GitHub - bulletphysics/bullet3: Bullet Physics SDK: real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning etc.
https://github.com/bulletphysics/bullet3

[2] [3] GitHub - Fuerdinger/cuda-particles: Simulate beautiful particle collisions with C++ and CUDA
https://github.com/Fuerdinger/cuda-particles

[4] Verlet integration - Wikipedia
https://en.wikipedia.org/wiki/Verlet_integration

[5] cs.cmu.edu
http://www.cs.cmu.edu/afs/cs/academic/class/15462-s15/www/project/p4/physics/p4physics.pdf

[6] [7] [8] [9] [10] Open-Source Electromagnetic Simulation: FDTD, FEM, MoM
https://www.epsilonforge.com/post/open-source-electromagnetics/

[11] C++ compile-time Maxwell's equations simulator | by Georgii Evtushenko | Medium
https://medium.com/@evtushenko.georgy/c-compile-time-maxwells-equations-simulator-65dc799c40b3

[12] [13] [14] [15] [16] C++ Black Hole Simulation Using OpenGL and Schwarzschild Geodesics | by Suyash Vishwakrma | Feb, 2026 | Medium
https://medium.com/@suyash.svish06/c-black-hole-simulation-using-opengl-and-schwarzschild-geodesics-14591dae61ba

[17] Split-step method - Wikipedia
https://en.wikipedia.org/wiki/Split-step_method

[18] QEngine | Quatomic
https://www.quatomic.com/qengine/

19 GitHub - QuEST-Kit/QuEST: A multithreaded, distributed, GPU-accelerated simulator of quantum computers
https://github.com/QuEST-Kit/QuEST

20 Sim 10: Numerically solving Schrodinger via Crank-Nicolson in C++
https://blog.c0nrad.io/posts/sim-10-crank-that-nicolson/

21 Metropolis–Hastings algorithm - Wikipedia
https://en.wikipedia.org/wiki/Metropolis%E2%80%93Hastings_algorithm

22 [2504.04465] GPU-based compressible lattice Boltzmann simulations on non-uniform grids using standard C++ parallelism: From best practices to aerodynamics, aeroacoustics and supersonic flow simulations
https://arxiv.org/abs/2504.04465

23 Maxwell–Boltzmann distribution - Wikipedia
https://en.wikipedia.org/wiki/Maxwell%E2%80%93Boltzmann_distribution

24 A Simple Particle Simulator in C++ and OpenGL - DEV Community
https://dev.to/yjdoc2/a-simple-particle-simulator-in-c-and-opengl-2p84