The main simulation loop works as follows:

- Active physicalized objects (i.e. those with simulation class 2) are grouped into separate islands by following their collider lists
- Their *Step* function is called, which advances their positions and orientations and checks for collisions. Steps are split into sub-steps if they exceed and object's step cap. Objects that move "fast enough" first do a continuous linear check along their velocity vector, and stop at the first contact. Note that if they initially intersect with something, this check will detect only *new* contacts along the way (for instance, if there's a triangle-triangle intersection, the next event will be one triangle's edge hitting the other triangle's edge). The main collision data comes from static intersection check with penetration depth
- Since collision detection changes collider lists, new islands are formed and the entities are requested to register all their contacts and constraints in a global list
- Contact "solver" is invoked for each island
- The process is repeated until all objects in the island complete their step, which can take several iterations due to step splitting
- Once all "purely physical" objects are finished, "living" and "independent" entities make their steps. However, it's possible for some of them to participate in the "purely physical" step if they are in the vicinity or are manually registered as colliders of some purely physical entities
- Islands and independent entities can be distributed among several worker threads

Constraint and contact solver consists of four parts. First, if the number of contacts is low enough, it attempts to use a direct matrix solver. If this doesn't yield a valid solution (which can happen, since it attempts to linearize solve a "linear complimentary" problem - "LCP"), iterative impulse solver is used (in the code it's referred to as "MC[microcontact] LCP solver"). In most cases this is the "main" solver. Then, if desired accuracy is still not reached, and the number of contacts is not too high, "LCP CG" solver is used. It makes several iterations of preconditioned conjugate gradient to drive velocities to 0 at "sticky" (non-separating) contacts. After each iteration it checks for negative normal velocities or adhesive normal impulses and moves violating contacts to the corresponding groups. The last iteration favors adhesive impulses over negative normal velocities. It treats each contact as having either infinite friction or 0 friction (this keeps the matrix symmetric and positive definite). A contact is assigned one of these types basing on the results of the MC LCP solver (that is, sticky contacts will remain sticky, and sliding contacts will assume that all tangential friction was already applied during the MC LCP stage). LCP CG solver does not try to enforce unprojection velocities for penetrating contacts; the last stage, "consecutive MINRES unprojector" is used for this purpose. It detects conflicting unprojection requests for each rigid body ("unprojection sandwiches"), and then attempts to find an unprojection order that avoid these conflicts. After that rigid bodies are assigned unprojection velocities one by one in that order (using MINRES solver), and these velocities are applied immediately, without affecting global velocities. All solvers are highly customizable in terms of iteration limits and thresholds (mostly via p_max_..., p_accuracy_.. cvars), and can thus be tweaked for different scenarios if necessary.

Wheeled vehicles are built on top of rigid bodies and use wheels to sample environment (by doing wheel geometry casts along suspension directions) and add suspension springs and tire friction impulses directly to the hull. Thus, tires are not modeled as separate rigid bodies. Then wheel contacts are registered for the global solver as special contacts.

Articulated bodies actually have two simulation modes. When they are alive, *Featherstone* solver is used on top of animation. Dead bodies use general rigid body solver with constraints by default, but *Featherstone* solver can be used as well. When a body is moving relatively slow, limbs are stepped independently, and the solver resolves joint drift errors (similar to the way it handles penetration depth). During fast movement bodies evolve as an articulated structure, thus avoiding limbs shifting far away when hit by strong impulses. Each limb has an individual sleep flag, and collisions for sleeping parts against inactive objects are not checked.

Particles (grenades, rockets) are treated in a simplified way. They do not have their time steps capped (since they should be ably to fly at high velocities without affecting performance) and use ray casts along their movement direction. Once they have bounced velocity below some threshold along the contact normal, they switch to sliding mode and attempt to follow the surface they are lying on while still shooting rays forward.

Ropes work as isolated entities. They can be attached to other objects with either both or just one end. If collision checks are enabled, upon each intersection they try to rotate intersecting segments to resolve penetrations, and then maintain safe distance. When ropes are not colliding, a "direct" (exact) solver is used, and an iterative one is used otherwise. There's also an optional "hard" length enforcement. When a rope is strained, it adds constraint to the bodies it's attached to, and this constraint is solved in the global rigid body solver (this constraint is more similar to a contact in that it can only pull the bodies, not push). Ropes can also work in a "dynamic subdivision" mode, where they add internal vertices to each segment to follow the collider's geometry more precisely. In this mode they register a special "rope" constraint in the global solver (it contains information about all rope's collision points).