Most of the interactions with physical entities will go through the functions `AddGeometry`, `SetParams`, `GetParams`, `GetStatus` and `Action`. Entities can have multiple geometries ('physicalized geometries') which can be added with a call to `AddGeometry` (there is a corresponding `RemoveGeometry` function as well). `SetParams` sets various parameters. These can always be queried later with `GetParams`. `GetStatus` requests the state of simulation parameters. The difference from `GetParams` is that `GetStatus` requests values that an entity is supposed to change during simulation, i.e. a simulation output, while `GetParams` queries parameters (although there's some overlap among them). `Action` makes an entity execute some action like having an impulse added. All these functions take structure pointers as parameters. When you want to issue a particular command, declare a corresponding structure somewhere (as a local variable, for example) and fill only the fields you need (most structures have more fields than you usually need to use in one call). The constructor of each structure fills all fields with a special value that tells the physics engine that this field was unused (the MARK_UNUSED macro can be used to do that explicitly and `is_unused` to check if the field is unused).

`AddGeometry` adds a physicalized geometry to an entity. Each geometry has the following properties:

- **id**: A unique part identifier within the bounds of the entity the geometry belongs to. Can be either specified explicitly or generated automatically (`AddGeometry` returns this value). id never changes when the parts array changes (if some parts from the middle are removed, for instance) but the internal parts index might.
- **position**, **orientation** and **uniform scaling** relative to the entity. Non-uniform scaling can be specified in an optional matrix parameter, but note that dynamic non-uniform scaling is not supported, so internally the physics will just clone the geometry and 'bake' non-uniform scale into it
- **mass**: Makes sense only for non-static objects; static objects assume infinite mass in all interactions). It is possible to specify either mass or density where the complementary value will be computed automatically (using formula mass = density*volume; *volume* is stored in the physicalized geometry structure and is scaled if the geometry is scaled).
- **surface_idx**: Used if neither IGeometry nor physicalized geometry have surface (material) identifiers.
- **flags** and **flagsCollider**: Whenever an entity checks collisions against other objects, it checks only parts that have a flag mask which intersects its current part's flagsCollider (i.e. `[CRYENGINE:current entity part].flagsCollider & [CRYENGINE:potential collider part].flags != 0`). There are 16 type bits that can be used (named *geom_colltype*) to represent some entity groups. Although it is not enforced, it is a good practice to keep these relationships symmetrical like this:
  `(entityA part_i).flags & (entityB part_j).flagsCollider -> (entityB part_j].flags & (entityA part_i).flagsCollider`
  In situations when it is known that collision checks are one-sided (i.e. entity A can check collisions against entity B but never vice versa), it is ok to not maintain this rule. Some flags are reserved for special collision groups, such as geom_colltype1 is equal to geom_colltype_players and geom_colltype2 is equal to geom_colltype_explosion (i.e. whenever explosion pressure is calculated, only parts with this flag are considered). There are also special flags for ray-tracing and buoyancy calculations, geom_colltype_ray and geom_floats respectively (it is possible however to specify different flags to be used for raytraycing). Collision Classes are an additional way of filtering interactions between entities.
- **minContactDist**: Minimum distance between contacts which this part of the entity might have with another part of another entity (note that contacts belonging to different parts are not checked for this). Can be left unused so that it will be initialized with a default value based on geometry size.
  Each part can have both geometry and proxy geometry. Geometry is used exclusively for raytracing and proxy geometry in all other cases. If no proxy geometry is specified, both geometries are set to be equal. The idea is to allow the raytracing to test against high-poly meshes without having to introduce changes to the part array layout.

Common Argument Structures

pe_params_pos
Sets position and orientation of the entity. One can use either offset/quaternion/scaling values directly or let the physics extract them from a 3x3 (orientation+scaling) or a 4x4 (orientation_scaling+offset) matrix. Physics use a right-to-left transformation order convention, with vectors being columns, i.e. vector_in_world = Matrix_Entity * Matrix_Entity_Parts * vector_in_geometry. All interface structures that support matrices can use either row-major or column-major matrix layout in memory (the latter is considered to be transposed, thus the corresponding member has T at the end of its name). Note that there is no per-entity scaling. Scaling is present only for parts so whenever a new scaling is set via pe_params_pos, it gets copied into each part and will override any previous individual scalings that the parts might have had. This structure also allows to set the simulation type manually. Normally, after the changes are made, entity bounds are recalculated and the entity gets re-registered in the collision hash grid although this can be postponed if *bRecalcBounds* is set to 0.

pe_params_bbox
Allows to force an entity's bounding box to a particular value, or query it (when used with `GetParams`). Normally, the bounding box is recalculated automatically basing on the geometries the entity has but setting it manually can be useful for entities without geometries (e.g. triggers) or placeholders. Note that if the entity has some geometries, it might recalculate its bounding box later, overriding these values. Bounding boxes are axis-aligned and in the world coordinate system.

pe_params_outer_entity
Allows to specify an outer entity for an entity. Whenever a box of interest (namely its center) is inside the entity with an outer entity, this outer entity is excluded from the set of potential colliders. This way it is possible to have a building exterior quickly culled away when the region of interest is inside the building's interior. Outer entities can be nested and an optional geometry to test for containment is supported.

pe_params_part
Sets or queries the entity part's properties. Part can be specified either using an internal part index or by its id.

pe_simulation_params
Sets simulation parameters for entities that can accept these parameters (currently these are all purely physical entities, ropes, and soft entities). *minEnergy* is equal to sleep speed squared. Note that damping and gravity can be specified independently for colliding and falling (i.e. when there are no contacts) states. An important member if simulation parameters is *maxAllowedStep*, which sets the maximum step the physical entity can make. If the current world step is larger than that, for this entity it will be split into several steps of maximum allowed duration and a remainder. For object groups ("simulation islands"), the minimum step cap will be used.

pe_params_buoyancy
The most common use is to set parameters of water/air physical areas. For other entities, it can be used to scale medium water density or resistance. For instance, if on object is detected to be in a water area with water density 1000 kg/m^3 and has density scale 0.01, bouncy effects will be computed as if water density was 1000*0.01 = 10kg/m^3. Scales can be set directly via members prefixed with 'k', or as pseudo-absolute values, in which case they'll be stored as scales relative to the default values, stored in the global water area. This scaling can be used to simulate non-watertight geometries, such as a metal cage that's represented by a simple box in the physics.

pe_action_impulse
Allows to add a one-time impulse to an entity. *impulse* is the impulse property (in N*s; impulse P will change object's velocity by P/mass). *point* is a point in world space where the impulse is applied. It is used to calculate the rotational effects of the impulse. Instead *of point*, momentum can be used to specify the rotational impulse explicitly. If neither point nor momentum are specified, the impulse is assumed to be applied to the center of mass of the object. *iAppl yTime* specifies the time when the impulse is applied. It is 2 ("after the next step") by default because if the object is lying on something, it is better to first give the solver an opportunity to reflect the impulse (it might even absorb it entirely) instead of trying to make a step with a potentially very high velocity first.

pe_action_add_constraint
Adds a constraint between two objects (currently only purely physical entities support this action). Points specify the constraint positions in world space (if the second point is not unused and is different from the first one, the solver will attempt to bring them together). Relative positions can be constrained to be either fixed, lie on the same line, on the same plane, or be fully independent, and relative rotations can be constrained in twist and bend directions. These directions correspond to rotation around x-axis and remaining rotation around a line on the yz-plane (it describes the tilt of the x axis) of a relative transformation between the two constraint coordinate frames attached to the affected bodies. Original position of the constraint frames are specified with *qfr ame* parameters and can be specified either in world or entity coordinate space (as indicated by the corresponding flag in *flags*). If one or both qframes are unused, they are considered to be an identity transformation in either the world or entity frame. Rotation limits are specified with the *xlimits* and *yzlimits* par ameters, with the element 0 being the minimum and element 1 the maximum (if the minimum is more than or equal to the maximum, the corresponding relative rotation is entirely prohibited).

For line positional constraints, x direction in the child's frame defines the direction of the line, and for plane constraints x direction defines the plane normal. Constraint frame if always owned by the heavier entity of the two constrained ones (with statics being considered unconditionally heavier than rigidbodies) and moves with it.

pConstraintEntity specifies an entity that represents the constraint. It can be a rope entity for constraints automatically created by strained ropes (in non-subdivided mode), or it can be a physical area (created via physical world's *AddArea*). In the latter case areas with *IGeometries* act as surface-based constraints and expect the rest of the constraint to be set as as plane constraint (which will have its attachment point continuously projected on the surface and normal aligned with the surface normal), and 3D spline areas act as line-based constraints and expect the rest of the constraint to be set as a line constraint (which will have its attachment point projected on the spline and x direction aligned with the spline direction). Note that area-based constraints still need some *pBuddy* to constrain to. If *pBuddy* is dynamic, the area (from *pConstraintEntity*) can be attached to it manually via another constraint, but it doesn't have to (and it's not assumed automatically).

When passed a *pe_action_add_constraint* pointer, `Action` returns a constraint identifier (which can later be used to remove the constraint). It will never be 0, so 0 indicates a failure.

pe_action_set_velocity
Sets velocity of an object directly. This is a very useful action for rigid bodies with infinite mass (represented as mass). It informs the physics system about the velocity that this body is moving with which will help the solver to ensure zero relative velocity with the objects it contacts (if velocity is not set and only position is changed, the engine will rely solely on penetrations to enforce the contacts). Note that velocity will not be automatically computed if position is set manually each frame. The body will continue moving with the specified velocity once it has been set.

pe_status_pos
Requests the current transformation (position, orientation, and scale) of an entity or its part. An alternative way is to use *pe_params_pos* with `GetParams`. If some matrix pointers are set, the engine will fill them with data in the corresponding format. Note that the *BBox* member in this structure is relative to entity's position, unlike *pe_params_bbox*.

pe_status_dymamics
Serves as a common way for retrieving an entity's movement parameters. Acceleration and angular acceleration are computed based on gravity and interactions with other objects. Note that any external impulses that might have been added to the entity are not taken into account since they are deemed to be instantaneous. *submergedFraction* is a fraction of the entity's volume that was under water during the last frame (naturally, only parts with geom_float flag are considered). *waterResistance* contains the maximum water resistance the entity encountered during one frame since the last time this status was requested (this means the accumulated value is cleared when the status is returned). This value can be useful for generating splash effects.

Living Entity-Specific Argument Structures

Living entities use cylinders or capsules as their bounding geometry. Normally the cylinders are hovering above the ground and the entity shoots a single ray down to detect if it is standing on something. This cylinder geometry always occupies the first part slot (it is created automatically). It is possible to add more geometries manually if necessary. Other entities however will process them when testing collisions against this entity. Living entities never change their orientation themselves - they are always set from outside. Normally, living entities are expected to rotate only around the z-axis but other orientations are supported with exception that collisions against livings entities always assume vertically oriented cylinders.

pe_player_dimensions (`GetParams`/`SetParams`)
Sets the dimensions of the living entity's bounding geometry. *heightPivot* specifies the z-coordinate of a point in the entity frame that is considered to be at the feet level (usually 0). *heightEye* is the z-coordinate of the camera attached to the entity. This camera does not affect entity movement, its sole purpose is to smooth out height changes that the entity undergoes (during walking on a highly bumpy surface, such as stairs, after dimensions change and during landing after a period of flying). The camera position can be requested via the *pe_status_living* structure. *sizeCollider* specifies the size of the cylinder (x is radius, z is half-height, y is unused). *heightColliders* is the cylinder's center z-coordinate. The head is an auxiliary sphere that is checked for collisions with objects above the cylinder. Head collisions don't affect movement but they make the camera position go down. *headRadius* is the radius of this sphere and *headHeight* is the z-coordinate of its center in the topmost state (i.e. when it doesn't touch anything).

pe_player_dynamics (`GetParams`/`SetParams`)
Sets a living entity's movement parameters. Living entities have their 'desired' (also called 'requested') movement velocity (set with *pe_action_move*) and they attempt to reach it. How fast that happens depends on the *kInertia* setting. The more this value is, the faster this velocity is reached. The default is 8 and 0 is a special case which means that the desired velocity will be reached instantly. *kAirControl* (0..1) specifies how strongly the requested velocity affects movement when the entity is flying (1 means that whenever a new requested velocity is set, it is copied to the actual movement velocity). *kAirResist ance* describes how fast velocity is damped during flying. *nodSpeed* (default 60) sets the strength of camera reaction to landings. *bSwimming* is a flag that tells that the entity is allowed to attempt to move in all directions (gravity might still pull it down though). If not set, the requested velocity will always be projected on the ground if the entity is not flying. Living entities have several threshold angles that specify maximum or minimum ground slopes for certain activities (*minSlideAngle*, *maxClimbAngle*, *maxJumpAngle* and *minFallAngle*). Note that if an entity's bounding cylinder collides with a sloped ground, the behavior is not governed by these slopes only. Setting *bNetwork* makes the entity allocate a much longer movement history array which might be required for synchronization (if not set, this array will be allocated the first time network-related actions are requested, such as performing a step back). Setting *bActi ve* to 0 puts the living entity to a special 'inactive' state where it does not check collisions with the environment and only moves with the requested velocity (other entities can still collide with it, though; note that this applies only to the entities of the same or higher simulation classes).

pe_action_move
Requests a movement from a living entity. *dir* is the requested velocity the entity will try to reach. If *iJump* is not 0, this velocity will not be projected on the ground and snapping to the ground will be turned off for a short period of time. If *iJump* is 1, the movement velocity is set to be equal to *dir* instantly. In case of 2, *dir* is added to it). *dt* is reserved for internal use.

pe_status_living
Returns the status of a living entity. *vel* is the velocity that is averaged from the entity's position change over several frames. *velUnconstrained* is the current movement velocity. It can be different from *vel* because in many cases when the entity bumps into an obstacle, it will restrict the actual movement but keep the movement velocity the same, so that if on the next frame the obstacle ends, no speed will be lost. If the entity is standing on something, *groun dHeight* and *groundSlope* will contain the point's z coordinate and normal, otherwise *bFlying* will be 1. Note that *pGroundCollider* is set only if the entity is standing on a non-static object. *camOffset* contains the current camera offset as a 3d vector in the entity frame (although only z coordinates actually changes in it). *bOnStairs* is a heuristic flag that indicates that the entity assumes that it is currently walking on stairs because of often and abrupt height changes.

Walking Rigid Entity-Specific Argument Structures

Walking Rigid entity is an implementation of Living Entity, using Rigid Entity as the base. It can use sweep-and-slide checks like livings, but can also maintain complex and persistent contacts with other physicalized objects like rigids. It supports some of the living interface structures for (limited) back ward compatibility, such as *pe_player_dimensions* (which will automatically create a collider geometry and a ground ray), and *pe_status_living*. As a rigid body, it's set to have infinite rotation inertia, i.e. the physics won't be able to rotate it (only move/translate). Like living entities, it needs "ground rays" to be able to enter "standing" mode, but unlike living it can have several such rays (only the closest one will be treated as a contact each frame, though). Leg contact is a special type of contact with customizable softness and friction. All other velocity changing functionality (such as everything related to "inertia") was moved out of the physics and the calling game code is expected to implement it in any way it needs. SampleRigidbodyActor component contains an implementation that replicates living entity behavior.

pe_params_sensors (`GetParams`/`SetParams`)
Sets up ground rays. *pOrigins* sets the origins (in the entity space), and *pDirections* - directions with length. It's important to set the rays so that they can touch the ground during normal walking (i.e. end lower than any collider geometry), and ideally start inside some geometry, so that that geometry can protect them from going under the ground (if the entire ray is under the ground / inside some geometry, it won't be able to detect a hit).

pe_params_walking_rigid (`GetParams`/`SetParams`)
Sets up friction and softness (along the leg direction) of the ground ray ("legs") contacts. Additionally, one can specify collision type- and mass threshold-based filtering for ground ray collisions (*minLegTestMass* and *legsColltype*). *velLegStick* sets how fast the ground must be moving away from the ray's bottom in order for the entity to lose ground contact and enter falling mode (normally you'd want some moderate value for walking on uneven terrain, particularly downwards, and 0 for lift-off during jumping).

Particle Entity-Specific Structures

pe_params_particle
Sets particle entity parameters. During movement, particles trace rays along their paths with the length *size*0.5* (since *size* stands for 'diameter' rather than 'radius') to check if they hit something. When they already lie or slide, they positions themselves at a distance of *thickness*0.5* from the surface (thus thin objects like shards of glass can be simulated). Particles can be set to have additional acceleration due to thrust of a lifting force (assuming that they have wings) with the parameters *accThrust* and *accLift* but these should never be used without specifying *kAirResistance*, otherwise particles will gain infinite velocity. Particles can optionally spin when in the air (toggled with flag *particle_no_spin*). Spinning is independent from linear motion of particles and is changed only after impacts or falling from surfaces. Particles can also align themselves with the direction of the movement (toggled with *particle_no_path_a lignment* flag) which is very useful for objects like rockets. That way the y-axis of the entity is aligned with the heading and the z-axis is set to be orthogonal to y and to point upward ('up' direction is considered to be opposite to particle's gravity). When moving along a surface, particles can either slide or roll. Rolling can be disabled with the flag *particle_no_roll* (it is automatically disabled on steep slopes). Note that rolling uses the particle material's friction as damping while rolling treats friction in a conventional way. When touching ground, particles align themselves so that their normal (defined in entity frame) is parallel to the surface normal. Particles can always keep the initial orientation as well (*particle_constant_orientation*) and stop completely after the first contact (*particle_single_contact*). *minBounceVel* specifies the lower velocity threshold after which the particle will not bounce even if the bounciness of the contact is more than 0.

Articulated Entity-Specific Structures

Articulated entities consist of linked rigid bodies. Loops in linking are not allowed and the connection structure should be a tree with a single root. A structural unit of an articulated entity is a rigid body with a joint that connects it to its parent (there can be only one such joint). When an articulated entity is used to simulate some body effects without interactions with environment, it uses the Featherstone method which is tweaked to tolerate strong impacts and stiff springs in complex body structures. In interactive mode it uses a common solver.

pe_params_joint
Creates a joint between two bodies in an articulated entity or changes (or queries when used with `GetParams`) parameters of an existing one. A joint is created between the two bodies specified in the *op* parameter at the pivot point (in the entity frame). Whenever a geometry is added to an articulated entity, it should use *pe_articgeomparams* to specify which body this geometry belongs to (in *idbody*). *idbody* can be any unique number and each body can have several geometries. There are no restrictions on the order in which joints are created but it is required that all bodies in the entity are connected before the simulation starts. Joints use Euler angles to define rotational limits. Flags that start with *angle0_* can be specified for each angle individually by shifting them left by the 0-based angle index (for instance, to lock the z-axis one should OR the flags with `angle0_locked<<2`). The coordinate frame of the child body is inherited from the first entity part (geometry) that was assigned to it. Joint angular limits are defined in a relative frame between the bodies that the joint connects. The frame of the child body can optionally be offset by specifying a child's orientation that corresponds to rotation angles (0,0,0) (using q0, pMtx0, or pMtx0T). This can help to get limits that can be robustly represented using Euler angles. A general rule regarding limits is to make upper and lower bounds at least 15-20 degrees apart (this depends on simulation settings though and on how high the velocity of this joint might be) and keep the y-axis limit in -90..90 degrees range (preferably within some safe margin from its ends). Note that in the parameter structure all angles are defined in radians. pe_params_joint uses 3d vectors to represent groups of 3 values that define some property for each angle. In addition to limits, each angle can have a spring that will pull the angle to 0 and a dashpot which will dampen the movement as the angle approaches its limit. For convenience, springs are specified in acceleration terms, i.e. the stiffness and damping can stay the same for joints that connect bodies with different masses (also, damping can be computed automatically to yield a critically damped spring by specifying *auto_kd* for the corresponding angle). *joint_no_gravity* makes the joint to be not affected by gravity (this can be useful if we assume that forces that hold the joint in its default position are just enough to counter the gravity) and *joint_isolated_accelerations* makes the joint use a special mode where it treats springs not like physical springs but rather like some guidelines on what the acceleration should be like (this mode is recommended for simulating effects on a skeleton). These two flags, as well as springs and dashpots, are currently only supported in Featherstone mode. Effective joint angles are always the sum of *q* and *qext*. If springs are activated, they attempt to drive *q* (but not *qext*) to 0. The idea behind this is to allow to set some pose from animation and then apply physical effects relative to it. In articulated entities collisions between parts are checked only for pairs that are explicitly specified in *pSelfCollidingParts* (thus this setting is per body or per joint rather than per part).

pe_params_articulated_body
Allows to set and query articulated entity simulation mode parameters. Articulated entities can either be attached to something or be free, as set by the *bGr ounded* flag (the latter is not supported in Featherstone mode). When grounded, the entity can either fetch dynamic parameters from the entity it is attached to (if *bInreritVel* is set; the entity is specified in *pHost*) or have them set in *a*, *wa*, *w* and *v* parameters. *bCollisionResp* effectively switches between Featherstone mode (0) and constraint mode (1). *bCheckCollisions* turns collision detection on and off. Currently it is supported only for constraint mode. *iSimType* specifies a simulation type which defines the way in which bodies that comprise the entity evolve. 0 means that joint pivots are always enforced exactly by projecting the movement of child bodies to a set of constrained directions and 1 means that bodies evolve independently and rely on the solver to enforce the joints. The second mode is not supported in Featherstone mode and in constraint mode it is turned on automatically if some bodies are moving fast enough (thus it makes sense to keep this value at 1 since it makes slow motion smoother). Articulated entities support a so called "lying mode" that is turned on when the number of contacts becomes more than a specified threshold (*nCollLyingMode*). Lying mode has a separate set of simulation parameters, such as gravity and damping. This feature was designed for ragdolls in order to help them look less sloppy when they hit the ground and to simulate the high damping of a human body in a simple way (this is achieved by setting gravity to a lower value and damping to higher than usual). Note that standard simulation parameters can still be different from freefall ones. When using constraint mode, articulated entities can try to represent hinge joints (i.e. rotational joints with only axis enabled) as two point-to-point constrains which can give better results in some cases. In order for this to happen, the entity should have the *bExpandHinges* parameter set (it internally propagates this value to *joint_expand_hinge* flags for all joints, so don't try to set it for joints manually).

Rope Entity-Specific Structures

Ropes are simulated as chains of connected equal-length sticks ("segments") with point masses. Each segment can individually collide with the environment. Ropes can tie two entities together, in which case they will add a constraint to those when fully strained and won't affect their movement when not. In order to collide with other objects (pushing them if necessary) in a strained state, the rope must use dynamic subdivision mode (set by *rope_s ubdivide_segs* flag). **pe_params_rope** specifies all the parameters a rope needs to start working (rope entities don't need any geometry). If initial point positions are not specified, the rope is assumed to be hanging down from its entity position and if they are, segments should have equal length, within some error margin. Note that ropes don't use materials to specify friction but rather an explicit friction value. If **pe_params_rope** is passed to `GetParams`, *pPoints* will be a pointer to the first vertex in an internal rope vertex structure and *iStride* will contain the size of it.

Soft Entity-Specific Structures

There are two types of soft entities - mesh-based (they use a soft constraint-like solver; typically cloth objects) and tetrahedral lattice-based (they use a spring solver; typically jelly-like objects). The longest edges of all triangles can optionally be discarded with the *sef_skip_longest_edges* flag. Currently collisions are handled at vertex level only (vertices have a customizable thickness though) and work best against primitive geometries rather than meshes.

pe_params_softbody
This is the main structure to set up a working soft entity (another one is *pe_simulation_params*). Here thickness is the collision size of vertices (they are therefore treated as spheres). If some edge differs from the original length by more than *maxSafeStep*, positional length enforcement kicks in. Spring damping is defined with *kdRatio* as a ratio to a critically damped value (overall damping from *pe_simulation_params* is also supported). Soft entities will react to wind if *airResistance* is not 0 (if wind is 0, having non-zero *airResistance* would mean that the entity will look like it is additionally damped - air resistance will attempt to even surface velocity with air velocity). Same will happen under water, only the parameters specified in *pe_params_buoyancy* will be used instead. Note that the Archimedean force that acts on vertices submerged in the water will depend on the entity's density which should be defined explicitly in *pe_simulation_params* (dependence will be same as for rigid bodies - the force will be 0 if *waterDensity* is equal to *density*). *collTypes* enables collisions with entities of a particular simulation type using *ent_* masks.

pe_action_attach_points
Can be used to attach some of a soft entity's vertices to another physical entity. *piVtx* specifies vertex indices and *points* specify attachment positions in world space (if unused, current vertex positions will become attachment points). It's also supported for ropes.