

This work is licensed under the Creative Commons  
Attribution-NonCommercial-ShareAlike 3.0 Unported License.  
To view a copy of this license, visit  
<http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to  
Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Game Physics Framework Specifications  
Copyright Daniel S. Buckstein

UML - descriptions only

```
-----
"cParticle2D"                // particle describing moving object in 2D
-----
-mPosition : vec2
-mVelocity : vec2
-mAcceleration : vec2
-mMomentum : vec2
-mForce : vec2
-mMass : float
-mMassInv : float
-mRotation : float
-mVelocityAng : float
-mAccelerationAng : float
-mMomentumAng : float
-mTorque : float
-mInertia : float
-mInertiaInv : float
// position of particle; integral of velocity, 2nd integral of acceleration
// velocity of particle; derivative of position, integral of acceleration
// acceleration of particle; 2nd derivative of position, derivative of velocity
// linear momentum of particle; integral of force, used in collision resolution
// total force applied to particle; derivative of momentum, used in Newton-2
// non-negative mass; zero for non-moving object
// non-negative mass reciprocal; zero for non-moving object
// rotation of particle (about Z axis only)
// angular velocity (about Z axis only)
// angular acceleration (about Z axis only)
// angular momentum (about Z axis only)
// torque ("angular force"); used in Newton-2 for angular acceleration
// moment of inertia ("angular mass distribution")
// moment of inertia reciprocal
-----
+cParticle2D()                // constructor; initialize all values to default
+fSet...(vec2) : vec2        // 2D vector member mutator
+fSet...(float) : float      // float member mutator
+fGet...() : vec2            // 2D vector member accessor
+fGet...() : float           // float member accessor
+fSetMass(float) : float     // validate and set mass and reciprocal
+fSetInertia(float) : float  // validate and set moment of inertia and recip.
-----

-----
"cParticle3D"                // particle describing moving object in 3D
-----
-mPosition : vec3
-mVelocity : vec3
-mAcceleration : vec3
-mMomentum : vec3
-mForce : vec3
```

```

67 -mMass : float
68 -mMassInv : float
69 -mRotation : vec4
70 -mVelocityAng : vec3
71 -mAccelerationAng : vec3
72 -mMomentumAng : vec3
73 -mTorque : vec3
74 -mInertia : mat3
75 -mInertiaInv : mat3
76 // position of particle; integral of velocity, 2nd integral of acceleration
77 // velocity of particle; derivative of position, integral of acceleration
78 // acceleration of particle; 2nd derivative of position, derivative of velocity
79 // linear momentum of particle; integral of force, used in collision resolution
80 // total force applied to particle; derivative of momentum, used in Newton-2
81 // non-negative mass; zero for non-moving object
82 // reciprocal of mass; zero for non-moving object
83 // rotation of particle (quaternion angle-axis encoding)
84 // angular velocity (unit axis, magnitude is angle)
85 // angular acceleration (unit axis, magnitude is angle)
86 // angular momentum (unit axis, magnitude is angle)
87 // torque ("angular force"); used in Newton-2 for angular acceleration
88 // moment of inertia tensor ("angular mass distribution", local-space matrix)
89 // moment of inertia tensor inverse (local-space matrix)
90 -----
91 +cParticle3D() // constructor; initialize all values to default
92 +fSet...(vec3) : vec3 // 3D vector member mutator
93 +fSet...(float) : float // float member mutator
94 +fGet...() : vec3 // 3D vector member accessor
95 +fGet...() : float // float member accessor
96 +fSetMass(float) : float // validate and set mass and reciprocal
97 +fSetInertia(mat3) : mat3 // set moment of inertia tensor and inverse
98 -----
99
100
101
102 -----
103 "scIntegrator" // static class for particle integration algorithms
104 -----
105 +sfIntegrateEuler(x : float, dx_dt : float, dt : float) : float
106 +sfIntegrateEuler2D(x : vec2, dx_dt : vec2, dt : float) : vec2
107 +sfIntegrateEuler3D(x : vec3, dx_dt : vec3, dt : float) : vec3
108 +sfIntegrateEuler4D(x : vec4, dx_dt : vec4, dt : float) : vec4
109 +sfIntegrateKinematic(x : float, dx_dt : float, d2x_dt2 : float, dt : float) : float
110 +sfIntegrateKinematic2D(x : vec2, dx_dt : vec2, d2x_dt2 : vec2, dt : float) : vec2
111 +sfIntegrateKinematic3D(x : vec3, dx_dt : vec3, d2x_dt2 : vec3, dt : float) : vec3
112 +sfIntegrateKinematic4D(x : vec4, dx_dt : vec4, d2x_dt2 : vec4, dt : float) : vec4
113 // Euler integration for scalar (value, derivative, differential)
114 // Euler integration for 2D vector (value, derivative, differential)
115 // Euler integration for 3D vector (value, derivative, differential)
116 // Euler integration for 4D vector (value, derivative, differential)
117 // kinematic integration for scalar (value, derivative, 2nd derivative, diff)
118 // kinematic integration for 2D vector (value, derivative, 2nd derivative, diff)
119 // kinematic integration for 3D vector (value, derivative, 2nd derivative, diff)
120 // kinematic integration for 4D vector (value, derivative, 2nd derivative, diff)
121
122 +sfIntegrateParticlePosition2D(p : cParticle2D, dt : float) : cParticle2D
123 +sfIntegrateParticleVelocity2D(p : cParticle2D, dt : float) : cParticle2D
124 +sfIntegrateParticleRotation2D(p : cParticle2D, dt : float) : cParticle2D
125 +sfIntegrateParticleVelocityAng2D(p : cParticle2D, dt : float) : cParticle2D
126 // integrate particle 2D position using preferred method
127 // integrate particle 2D velocity using preferred method
128 // integrate particle scalar rotation using preferred method
129 // integrate particle 2D angular velocity using preferred method
130
131 +sfIntegrateParticlePosition3D(p : cParticle3D, dt : float) : cParticle3D
132 +sfIntegrateParticleVelocity3D(p : cParticle3D, dt : float) : cParticle3D

```

```

133 +sfIntegrateParticleRotation3D(p : cParticle3D, dt : float) : cParticle3D
134 +sfIntegrateParticleVelocityAng3D(p : cParticle3D, dt : float) : cParticle3D
135 // integrate particle 3D position using preferred method
136 // integrate particle 3D velocity using preferred method
137 // integrate particle quaternion rotation using preferred method
138 // integrate particle 3D angular velocity using preferred method
139
140 +sfCalculateRotationDerivative3D(rotation : vec4, velocityAng : vec3) : vec4
141 // calculate quaternion derivative for 3D rotation (half ang-velocity x rotation)
142 -----
143
144
145
146 -----
147 "scForceGenerator"          // static class for force generation algorithms
148 -----
149 +sfGenerateForceGravity2D(
150     up_world : vec2, coefficient_gravity : float, mass_particle : float) : vec2
151 +sfGenerateForceNormal2D(
152     fGravity : vec2, normal_surface : vec2) : vec2
153 +sfGenerateForceDrag2D(
154     velocity_particle : vec2, velocity_fluid : vec2, density_fluid : float,
155     crossSectionArea_object : float, coefficient : float) : vec2
156 +sfGenerateForceFrictionStatic2D(
157     fNormal : vec2, fOpposing : vec2, coefficient_static : float) : vec2
158 +sfGenerateForceFrictionKinetic2D(
159     fNormal : vec2, fOpposing : vec2, coefficient_kinetic : float,
160     velocity_particle : vec2) : vec2
161 +sfGenerateForceFriction2D(
162     fNormal : vec2, fOpposing : vec2, coefficient_static : float,
163     coefficient_kinetic : float, velocity_particle : vec2) : vec2
164 +sfGenerateForceSpring2D(
165     position_particle : vec2, position_anchor : vec2, restingLength_spring : float,
166     coefficient_stiffness : float) : vec2
167 +sfGenerateForceSpringDamped2D(
168     position_particle : vec2, position_anchor : vec2, restingLength_spring : float,
169     coefficient_stiffness : float, coefficient_damping : float,
170     mass_particle : float, velocity_particle : vec2) : vec2
171 // NOTE: forces described in the books are typically scalar quantities,
172 // which represent the magnitudes of the force vectors
173
174 +sfGenerateForceGravity3D(
175     up_world : vec3, coefficient_gravity : float, mass_particle : float) : vec3
176 +sfGenerateForceNormal3D(
177     fGravity : vec3, normal_surface : vec3) : vec3
178 +sfGenerateForceDrag3D(
179     velocity_particle : vec3, velocity_fluid : vec3, density_fluid : float,
180     crossSectionArea_object : float, coefficient : float) : vec3
181 +sfGenerateForceFrictionStatic3D(
182     fNormal : vec3, fOpposing : vec3, coefficient_static : float) : vec3
183 +sfGenerateForceFrictionKinetic3D(
184     fNormal : vec3, fOpposing : vec3, coefficient_kinetic : float,
185     velocity_particle : vec3) : vec3
186 +sfGenerateForceFriction3D(
187     fNormal : vec3, fOpposing : vec3, coefficient_static : float,
188     coefficient_kinetic : float, velocity_particle : vec3) : vec3
189 +sfGenerateForceSpring3D(
190     position_particle : vec3, position_anchor : vec3, restingLength_spring : float,
191     coefficient_stiffness : float) : vec3
192 +sfGenerateForceSpringDamped3D(
193     position_particle : vec3, position_anchor : vec3, restingLength_spring : float,
194     coefficient_stiffness : float, coefficient_damping : float,
195     mass_particle : float, velocity_particle : vec3) : vec3
196 // NOTE: forces described in the books are typically scalar quantities,
197 // which represent the magnitudes of the force vectors
198

```

```

199 +sfGenerateTorque2D(
200     centerOfMass_world : vec2, pointOfForce_world : vec2,
201     force_world : vec2) : float
202 +sfGenerateTorque3D(
203     centerOfMass_world : vec3, pointOfForce_world : vec3,
204     force_world : vec3) : vec3
205
206 +sfResetForce2D(p : cParticle2D) : cParticle2D
207 +sfApplyForce2D(p : cParticle2D, f : vec2) : cParticle2D
208 +sfConvertForce2D(p : cParticle2D) : cParticle2D
209 +sfResetTorque2D(p : cParticle2D) : cParticle2D
210 +sfApplyTorque2D(p : cParticle2D, t : float) : cParticle2D
211 +sfConvertTorque2D(p : cParticle2D) : cParticle2D
212 // set particle's force to zero
213 // add force to particle's total
214 // use Newton-2 to calculate 2D acceleration (force / mass)
215 // set particle's torque to zero
216 // add torque to particle's total
217 // use Newton-2 to calculate 2D angular acceleration (torque / inertia)
218
219 +sfResetForce3D(p : cParticle3D) : cParticle3D
220 +sfApplyForce3D(p : cParticle3D, f : vec3) : cParticle3D
221 +sfConvertForce3D(p : cParticle3D) : cParticle3D
222 +sfResetTorque3D(p : cParticle3D) : cParticle3D
223 +sfApplyTorque3D(p : cParticle3D, t : vec3) : cParticle3D
224 +sfConvertTorque3D(p : cParticle3D) : cParticle3D
225 // set particle's force to zero
226 // add force to particle's total
227 // use Newton-2 to calculate 3D acceleration (force / mass)
228 // set particle's torque to zero
229 // add torque to particle's total
230 // use Newton-2 to calculate 3D angular acceleration (torque x world tensor inverse)
231
232 +sfCalculateInertiaTensorWorld(
233     transform : mat3, inertia_local : mat3, transformInv : mat3) : mat3
234 // calculate world-space inertia tensor (product of inputs)
235 +sfCalculateInertiaInvTensorWorld(
236     transform : mat3, inertiaInv_local : mat3, transformInv : mat3) : mat3
237 // calculate world-space inertia inverse tensor (product of inputs)
238 -----
239
240
241
242 -----
243 "cCollisionHull2D"           // base class for 2D collision hull
244 -----
245 -mpParticle : cParticle2D    // pointer/reference to target particle
246 -mType : int                 // enumerated type of collider
247 -mTransform : mat3           // rigid transformation matrix
248 -mTransformInv : mat3        // rigid transformation matrix inverse
249 -----
250 #cCollisionHull2D()           // constructor; initialize all values to default
251 +fSet...(...) : ...           // mutators
252 +fGet...(...) : ...           // accessors
253 +fUpdateTransform() : mat3    // update transform and inverse using particle data
254 -----
255
256
257
258 -----
259 "cCollisionHullCircle2D"     // class for 2D circle collision hull
260 -----
261 -mRadius : float             // non-negative radius
262 -mRadiusSq : float           // squared radius for optimization
263 -----
264 +cCollisionHullCircle2D()     // constructor; initialize all values to default

```

```

265 +fSet...(...) : ...          // mutators
266 +fGet...(...) : ...          // accessors
267 -----
268
269
270
271 -----
272 "cCollisionHullAABB2D"        // class for 2D strictly axis-aligned box
273 -----
274 -mSize : vec2                 // non-negative dimensions
275 -mSizeHalf : vec2             // half dimensions for optimization
276 -mCornerLocal : vec2[4]       // array of corners in local-space
277 -mCornerWorld : vec2[4]       // array of corners in world-space
278 -----
279 +cCollisionHullAABB2D()        // constructor; initialize all values to default
280 +fSet...(...) : ...           // mutators
281 +fGet...(...) : ...           // accessors
282 +fUpdateCornersWorld()        // use transform to convert local corners to world
283 -----
284
285
286
287 -----
288 "cCollisionHullBox2D"         // class for 2D box, axis-aligned or not
289 -----
290 -mSize : vec2                 // non-negative dimensions
291 -mSizeHalf : vec2             // half dimensions for optimization
292 -mCornerLocal : vec2[4]       // array of corners in local-space
293 -mCornerWorld : vec2[4]       // array of corners in world-space
294 -----
295 +cCollisionHullBox2D()        // constructor; initialize all values to default
296 +fSet...(...) : ...           // mutators
297 +fGet...(...) : ...           // accessors
298 +fUpdateCornersWorld()        // use transform to convert local corners to world
299 -----
300
301
302
303 -----
304 "cCollisionContact2D"         // class for 2D collision contact description
305 -----
306 +mPoint : vec2                // contact location
307 +mNormal : vec2               // contact normal
308 +mDepth : float               // contact depth
309 -----
310
311
312
313 -----
314 "cCollision2D"                // class for 2D collision description
315 -----
316 +mpHull0 : cCollisionHull2D    // pointer/reference to first hull involved
317 +mpHull1 : cCollisionHull2D    // pointer/reference to second hull involved
318 +mContact : cCollisionContact2D[2] // array of contact descriptors
319 -----
320
321
322
323 -----
324 "cCollisionManager2D"         // class for 2D collision detection and resolution
325 -----
326 -mHull : cCollisionHull2D[]    // array of hulls to manage
327 -mCollision : cCollision2D[]   // array of collisions to manage
328 -----
329 +fAddHull(hull : cCollisionHull2D) : cCollisionHull2D // add hull to list
330 +fRemoveHull(hull : cCollisionHull2D) : cCollisionHull2D // remove hull

```

```

331 +fClearHulls() : int // clear hull list
332 +fClearCollisions() : int // clear collisions
333 +fDetectCollisions() : int // detect collisions
334 +fResolveCollisions() : int // resolve detected
335
336 +sfCollisionTest(collision_out : cCollision2D,
337     hullA : cCollisionHull2D, hullB : cCollisionHull2D) : bool
338 +sfCollisionTest_..._...(collision_out : cCollision2D,
339     hullA : cCollisionHull...2D, hullB : cCollisionHull...2D) : bool
340 // test two hulls of specific types (ellipsis replaced with hull type)
341 // example tests: _CIRCLE_CIRCLE; _CIRCLE_AABB/_AABB_CIRCLE;
342 // _CIRCLE_BOX/_BOX_CIRCLE; _AABB_AABB; _AABB_BOX/_BOX_AABB; _BOX_BOX;
343 -----
344
345
346
347 -----
348 "cCollisionHull3D" // base class for 3D collision hull
349 -----
350 -mpParticle : cParticle3D // pointer/reference to target particle
351 -mType : int // enumerated type of collider
352 -mTransform : mat4 // rigid transformation matrix
353 -mTransformInv : mat4 // rigid transformation matrix inverse
354 -----
355 #cCollisionHull3D() // constructor; initialize all values to default
356 +fSet...(...) : ... // mutators
357 +fGet...(...) : ... // accessors
358 +fUpdateTransform() : mat4 // update transform and inverse using particle data
359 -----
360
361
362
363 -----
364 "cCollisionHullSphere3D" // class for 3D sphere collision hull
365 -----
366 -mRadius : float // non-negative radius
367 -mRadiusSq : float // squared radius for optimization
368 -----
369 +cCollisionHullSphere3D() // constructor; initialize all values to default
370 +fSet...(...) : ... // mutators
371 +fGet...(...) : ... // accessors
372 -----
373
374
375
376 -----
377 "cCollisionHullAABB3D" // class for 3D strictly axis-aligned box
378 -----
379 -mSize : vec3 // non-negative dimensions
380 -mSizeHalf : vec3 // half dimensions for optimization
381 -mCornerLocal : vec3[8] // array of corners in local-space
382 -mCornerWorld : vec3[8] // array of corners in world-space
383 -----
384 +cCollisionHullAABB3D() // constructor; initialize all values to default
385 +fSet...(...) : ... // mutators
386 +fGet...(...) : ... // accessors
387 +fUpdateCornersWorld() // use transform to convert local corners to world
388 -----
389
390
391
392 -----
393 "cCollisionHullBox3D" // class for 3D box, axis-aligned or not
394 -----
395 -mSize : vec3 // non-negative dimensions
396 -mSizeHalf : vec3 // half dimensions for optimization

```

```

397 -mCornerLocal : vec3[8]      // array of corners in local-space
398 -mCornerWorld : vec3[8]     // array of corners in world-space
399 -----
400 +cCollisionHullBox3D()        // constructor; initialize all values to default
401 +fSet...(...) : ...          // mutators
402 +fGet...(...) : ...          // accessors
403 +fUpdateCornersWorld()       // use transform to convert local corners to world
404 -----
405
406
407
408 -----
409 "cCollisionContact3D"         // class for 3D collision contact description
410 -----
411 +mPoint : vec3                // contact location
412 +mNormal : vec3               // contact normal
413 +mDepth : float               // contact depth
414 -----
415
416
417
418 -----
419 "cCollision3D"                // class for 3D collision description
420 -----
421 +mpHull0 : cCollisionHull3D    // pointer/reference to first hull involved
422 +mpHull1 : cCollisionHull3D    // pointer/reference to second hull involved
423 +mContact : cCollisionContact3D[4] // array of contact descriptors
424 -----
425
426
427
428 -----
429 "cCollisionManager3D"         // class for 3D collision detection and resolution
430 -----
431 -mHull : cCollisionHull3D[]    // array of hulls to manage
432 -mCollision : cCollision3D[]   // array of collisions to manage
433 -----
434 +fAddHull(hull : cCollisionHull3D) : cCollisionHull3D // add hull to list
435 +fRemoveHull(hull : cCollisionHull3D) : cCollisionHull3D // remove hull
436 +fClearHulls() : int           // clear hull list
437 +fClearCollisions() : int       // clear collisions
438 +fDetectCollisions() : int      // detect collisions
439 +fResolveCollisions() : int     // resolve detected
440
441 +sfCollisionTest(collision_out : cCollision3D,
442     hullA : cCollisionHull3D, hullB : cCollisionHull3D) : bool
443 +sfCollisionTest_..._(collision_out : cCollision3D,
444     hullA : cCollisionHull...3D, hullB : cCollisionHull...3D) : bool
445 // test two hulls of specific types (ellipsis replaced with hull type)
446 // example tests: _SPHERE_SPHERE; _SPHERE_AABB/_AABB_SPHERE;
447 // _SPHERE_BOX/_BOX_SPHERE; _AABB_AABB; _AABB_BOX/_BOX_AABB; _BOX_BOX;
448 -----
449
450

```