Game Physics Framework Specifications

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

```
-mMass : float
-mMassInv : float
-mRotation : vec4
-mVelocityAng : vec3
-mAccelerationAng : vec3
-mMomentumAng : vec3
-mTorque : vec3
-mInertia : mat3
-mInertiaInv : mat3
// 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
// reciprocal of mass; zero for non-moving object
// rotation of particle (quaternion angle-axis encoding)
// angular velocity (unit axis, magnitude is angle)
// angular acceleration (unit axis, magnitude is angle)
// angular momentum (unit axis, magnitude is angle)
// torque ("angular force"); used in Newton-2 for angular acceleration
// moment of inertia tensor ("angular mass distribution", local-space matrix)
// moment of inertia tensor inverse (local-space matrix)
-------------------------------------------------------------------------------
+cParticle3D()              // constructor; initialize all values to default
+fSet...(vec3) : vec3       // 3D vector member mutator
+fSet...(float) : float     // float member mutator
+fGet...() : vec3           // 3D vector member accessor
+fGet...() : float          // float member accessor
+fSetMass(float) : float    // validate and set mass and reciprocal
+fSetInertia(mat3) : mat3   // set moment of inertia tensor and inverse
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"scIntegrator"             // static class for particle integration algorithms
-------------------------------------------------------------------------------
+sfIntegrateEuler(x : float, dx_dt : float, dt : float) : float
+sfIntegrateEuler2D(x : vec2, dx_dt : vec2, dt : float) : vec2
+sfIntegrateEuler3D(x : vec3, dx_dt : vec3, dt : float) : vec3
+sfIntegrateEuler4D(x : vec4, dx_dt : vec4, dt : float) : vec4
+sfIntegrateKinematic(x : float, dx_dt : float, d2x_dt2 : float, dt : float) : float
+sfIntegrateKinematic2D(x : vec2, dx_dt : vec2, d2x_dt2 : vec2, dt : float) : vec2
+sfIntegrateKinematic3D(x : vec3, dx_dt : vec3, d2x_dt2 : vec3, dt : float) : vec3
+sfIntegrateKinematic4D(x : vec4, dx_dt : vec4, d2x_dt2 : vec4, dt : float) : vec4
// Euler integration for scalar (value, derivative, differential)
// Euler integration for 2D vector (value, derivative, differential)
// Euler integration for 3D vector (value, derivative, differential)
// Euler integration for 4D vector (value, derivative, differential)
// kinematic integration for scalar (value, derivative, 2nd derivative, diff)
// kinematic integration for 2D vector (value, derivative, 2nd derivative, diff)
// kinematic integration for 3D vector (value, derivative, 2nd derivative, diff)
// kinematic integration for 4D vector (value, derivative, 2nd derivative, diff)

+sfIntegrateParticlePosition2D(p : cParticle2D, dt : float) : cParticle2D
+sfIntegrateParticleVelocity2D(p : cParticle2D, dt : float) : cParticle2D
+sfIntegrateParticleRotation2D(p : cParticle2D, dt : float) : cParticle2D
+sfIntegrateParticleVelocityAng2D(p : cParticle2D, dt : float) : cParticle2D
// integrate particle 2D position using preferred method
// integrate particle 2D velocity using preferred method
// integrate particle scalar rotation using preferred method
// integrate particle 2D angular velocity using preferred method

+sfIntegrateParticlePosition3D(p : cParticle3D, dt : float) : cParticle3D
+sfIntegrateParticleVelocity3D(p : cParticle3D, dt : float) : cParticle3D
```

```
+sfIntegrateParticleRotation3D(p : cParticle3D, dt : float) : cParticle3D
+sfIntegrateParticleVelocityAng3D(p : cParticle3D, dt : float) : cParticle3D
// integrate particle 3D position using preferred method
// integrate particle 3D velocity using preferred method
// integrate particle quaternion rotation using preferred method
// integrate particle 3D angular velocity using preferred method

+sfCalculateRotationDerivative3D(rotation : vec4, velocityAng : vec3) : vec4
// calculate quaternion derivative for 3D rotation (half ang-velocity x rotation)
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"scForceGenerator"          // static class for force generation algorithms
-------------------------------------------------------------------------------
+sfGenerateForceGravity2D(
    up_world : vec2, coefficient_gravity : float, mass_particle : float) : vec2
+sfGenerateForceNormal2D(
    fGravity : vec2, normal_surface : vec2) : vec2
+sfGenerateForceDrag2D(
    velocity_particle : vec2, velocity_fluid : vec2, density_fluid : float,
    crossSectionArea_object : float, coefficient : float) : vec2
+sfGenerateForceFrictionStatic2D(
    fNormal : vec2, fOpposing : vec2, coefficient_static : float) : vec2
+sfGenerateForceFrictionKinetic2D(
    fNormal : vec2, fOpposing : vec2, coefficient_kinetic : float,
    velocity_particle : vec2) : vec2
+sfGenerateForceFriction2D(
    fNormal : vec2, fOpposing : vec2, coefficient_static : float,
    coefficient_kinetic : float, velocity_particle : vec2) : vec2
+sfGenerateForceSpring2D(
    position_particle : vec2, position_anchor : vec2, restingLength_spring : float,
    coefficient_stiffness : float) : vec2
+sfGenerateForceSpringDamped2D(
    position_particle : vec2, position_anchor : vec2, restingLength_spring : float,
    coefficient_stiffness : float, coefficient_damping : float,
    mass_particle : float, velocity_particle : vec2) : vec2
// NOTE: forces described in the books are typically scalar quantities,
//  which represent the magnitudes of the force vectors

+sfGenerateForceGravity3D(
    up_world : vec3, coefficient_gravity : float, mass_particle : float) : vec3
+sfGenerateForceNormal3D(
    fGravity : vec3, normal_surface : vec3) : vec3
+sfGenerateForceDrag3D(
    velocity_particle : vec3, velocity_fluid : vec3, density_fluid : float,
    crossSectionArea_object : float, coefficient : float) : vec3
+sfGenerateForceFrictionStatic3D(
    fNormal : vec3, fOpposing : vec3, coefficient_static : float) : vec3
+sfGenerateForceFrictionKinetic3D(
    fNormal : vec3, fOpposing : vec3, coefficient_kinetic : float,
    velocity_particle : vec3) : vec3
+sfGenerateForceFriction3D(
    fNormal : vec3, fOpposing : vec3, coefficient_static : float,
    coefficient_kinetic : float, velocity_particle : vec3) : vec3
+sfGenerateForceSpring3D(
    position_particle : vec3, position_anchor : vec3, restingLength_spring : float,
    coefficient_stiffness : float) : vec3
+sfGenerateForceSpringDamped3D(
    position_particle : vec3, position_anchor : vec3, restingLength_spring : float,
    coefficient_stiffness : float, coefficient_damping : float,
    mass_particle : float, velocity_particle : vec3) : vec3
// NOTE: forces described in the books are typically scalar quantities,
//  which represent the magnitudes of the force vectors

```

```
+sfGenerateTorque2D(
    centerOfMass_world : vec2, pointOfForce_world : vec2,
    force_world : vec2) : float
+sfGenerateTorque3D(
    centerOfMass_world : vec3, pointOfForce_world : vec3,
    force_world : vec3) : vec3

+sfResetForce2D(p : cParticle2D) : cParticle2D
+sfApplyForce2D(p : cParticle2D, f : vec2) : cParticle2D
+sfConvertForce2D(p : cParticle2D) : cParticle2D
+sfResetTorque2D(p : cParticle2D) : cParticle2D
+sfApplyTorque2D(p : cParticle2D, t : float) : cParticle2D
+sfConvertTorque2D(p : cParticle2D) : cParticle2D
// set particle's force to zero
// add force to particle's total
// use Newton-2 to calculate 2D acceleration (force / mass)
// set particle's torque to zero
// add torque to particle's total
// use Newton-2 to calculate 2D angular acceleration (torque / inertia)

+sfResetForce3D(p : cParticle3D) : cParticle3D
+sfApplyForce3D(p : cParticle3D, f : vec3) : cParticle3D
+sfConvertForce3D(p : cParticle3D) : cParticle3D
+sfResetTorque3D(p : cParticle3D) : cParticle3D
+sfApplyTorque3D(p : cParticle3D, t : vec3) : cParticle3D
+sfConvertTorque3D(p : cParticle3D) : cParticle3D
// set particle's force to zero
// add force to particle's total
// use Newton-2 to calculate 3D acceleration (force / mass)
// set particle's torque to zero
// add torque to particle's total
// use Newton-2 to calculate 3D angular acceleration (torque x world tensor inverse)

+sfCalculateInertiaTensorWorld(
    transform : mat3, inertia_local : mat3, transformInv : mat3) : mat3
// calculate world-space inertia tensor (product of inputs)
+sfCalculateInertiaInvTensorWorld(
    transform : mat3, inertiaInv_local : mat3, transformInv : mat3) : mat3
// calculate world-space inertia inverse tensor (product of inputs)
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollisionHull2D"          // base class for 2D collision hull
-------------------------------------------------------------------------------
-mpParticle : cParticle2D   // pointer/reference to target particle
-mType : int                // enumerated type of collider
-mTransform : mat3          // rigid transformation matrix
-mTransformInv : mat3       // rigid transformation matrix inverse
-------------------------------------------------------------------------------
#cCollisionHull2D()         // constructor; initialize all values to default
+fSet...(...) : ...         // mutators
+fGet...(...) : ...         // accessors
+fUpdateTransform() : mat3  // update transform and inverse using particle data
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollisionHullCircle2D"    // class for 2D circle collision hull
-------------------------------------------------------------------------------
-mRadius : float            // non-negative radius
-mRadiusSq : float          // squared radius for optimization
-------------------------------------------------------------------------------
+cCollisionHullCircle2D()   // constructor; initialize all values to default
```

```
+fSet...(...) : ...         // mutators
+fGet...(...) : ...         // accessors
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollisionHullAABB2D"      // class for 2D strictly axis-aligned box
-------------------------------------------------------------------------------
-mSize : vec2               // non-negative dimensions
-mSizeHalf : vec2           // half dimensions for optimization
-mCornerLocal : vec2[4]     // array of corners in local-space
-mCornerWorld : vec2[4]     // array of corners in world-space
-------------------------------------------------------------------------------
+cCollisionHullAABB2D()     // constructor; initialize all values to default
+fSet...(...) : ...         // mutators
+fGet...(...) : ...         // accessors
+fUpdateCornersWorld()      // use transform to convert local corners to world
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollisionHullBox2D"       // class for 2D box, axis-aligned or not
-------------------------------------------------------------------------------
-mSize : vec2               // non-negative dimensions
-mSizeHalf : vec2           // half dimensions for optimization
-mCornerLocal : vec2[4]     // array of corners in local-space
-mCornerWorld : vec2[4]     // array of corners in world-space
-------------------------------------------------------------------------------
+cCollisionHullBox2D()      // constructor; initialize all values to default
+fSet...(...) : ...         // mutators
+fGet...(...) : ...         // accessors
+fUpdateCornersWorld()      // use transform to convert local corners to world
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollisionContact2D"       // class for 2D collision contact description
-------------------------------------------------------------------------------
+mPoint : vec2      // contact location
+mNormal : vec2     // contact normal
+mDepth : float     // contact depth
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollision2D"              // class for 2D collision description
-------------------------------------------------------------------------------
+mpHull0 : cCollisionHull2D         // pointer/reference to first hull involved
+mpHull1 : cCollisionHull2D         // pointer/reference to second hull involved
+mContact : cCollisionContact2D[2]  // array of contact descriptors
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollisionManager2D"       // class for 2D collision detection and resolution
-------------------------------------------------------------------------------
-mHull : cCollisionHull2D[]     // array of hulls to manage
-mCollision : cCollision2D[]    // array of collisions to manage
-------------------------------------------------------------------------------
+fAddHull(hull : cCollisionHull2D) : cCollisionHull2D       // add hull to list
+fRemoveHull(hull : cCollisionHull2D) : cCollisionHull2D    // remove hull
```

```
+fClearHulls() : int                                        // clear hull list
+fClearCollisions() : int                                   // clear collisions
+fDetectCollisions() : int                                  // detect collisions
+fResolveCollisions() : int                                 // resolve detected

+sfCollisionTest(collision_out : cCollision2D,
    hullA : cCollisionHull2D, hullB : cCollisionHull2D) : bool
+sfCollisionTest_..._...(collision_out : cCollision2D,
    hullA : cCollisionHull...2D, hullB : cCollisionHull...2D) : bool
// test two hulls of specific types (ellipsis replaced with hull type)
// example tests: _CIRCLE_CIRCLE; _CIRCLE_AABB/_AABB_CIRCLE;
//  _CIRCLE_BOX/_BOX_CIRCLE; _AABB_AABB; _AABB_BOX/_BOX_AABB; _BOX_BOX;
-------------------------------------------------------------------------------




-------------------------------------------------------------------------------
"cCollisionHull3D"          // base class for 3D collision hull
-------------------------------------------------------------------------------
-mpParticle : cParticle3D   // pointer/reference to target particle
-mType : int                // enumerated type of collider
-mTransform : mat4          // rigid transformation matrix
-mTransformInv : mat4       // rigid transformation matrix inverse
-------------------------------------------------------------------------------
#cCollisionHull3D()         // constructor; initialize all values to default
+fSet...(...) : ...         // mutators
+fGet...(...) : ...         // accessors
+fUpdateTransform() : mat4  // update transform and inverse using particle data
-------------------------------------------------------------------------------




-------------------------------------------------------------------------------
"cCollisionHullSphere3D"    // class for 3D sphere collision hull
-------------------------------------------------------------------------------
-mRadius : float            // non-negative radius
-mRadiusSq : float          // squared radius for optimization
-------------------------------------------------------------------------------
+cCollisionHullSphere3D()   // constructor; initialize all values to default
+fSet...(...) : ...         // mutators
+fGet...(...) : ...         // accessors
-------------------------------------------------------------------------------




-------------------------------------------------------------------------------
"cCollisionHullAABB3D"      // class for 3D strictly axis-aligned box
-------------------------------------------------------------------------------
-mSize : vec3               // non-negative dimensions
-mSizeHalf : vec3           // half dimensions for optimization
-mCornerLocal : vec3[8]     // array of corners in local-space
-mCornerWorld : vec3[8]     // array of corners in world-space
-------------------------------------------------------------------------------
+cCollisionHullAABB3D()     // constructor; initialize all values to default
+fSet...(...) : ...         // mutators
+fGet...(...) : ...         // accessors
+fUpdateCornersWorld()      // use transform to convert local corners to world
-------------------------------------------------------------------------------




-------------------------------------------------------------------------------
"cCollisionHullBox3D"       // class for 3D box, axis-aligned or not
-------------------------------------------------------------------------------
-mSize : vec3               // non-negative dimensions
-mSizeHalf : vec3           // half dimensions for optimization
```

```
-mCornerLocal : vec3[8]     // array of corners in local-space
-mCornerWorld : vec3[8]     // array of corners in world-space
-------------------------------------------------------------------------------
+cCollisionHullBox3D()      // constructor; initialize all values to default
+fSet...(...) : ...         // mutators
+fGet...(...) : ...         // accessors
+fUpdateCornersWorld()      // use transform to convert local corners to world
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollisionContact3D"      // class for 3D collision contact description
-------------------------------------------------------------------------------
+mPoint : vec3      // contact location
+mNormal : vec3     // contact normal
+mDepth : float     // contact depth
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollision3D"             // class for 3D collision description
-------------------------------------------------------------------------------
+mpHull0 : cCollisionHull3D         // pointer/reference to first hull involved
+mpHull1 : cCollisionHull3D         // pointer/reference to second hull involved
+mContact : cCollisionContact3D[4]  // array of contact descriptors
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
"cCollisionManager3D"      // class for 3D collision detection and resolution
-------------------------------------------------------------------------------
-mHull : cCollisionHull3D[]     // array of hulls to manage
-mCollision : cCollision3D[]    // array of collisions to manage
-------------------------------------------------------------------------------
+fAddHull(hull : cCollisionHull3D) : cCollisionHull3D      // add hull to list
+fRemoveHull(hull : cCollisionHull3D) : cCollisionHull3D   // remove hull
+fClearHulls() : int                                       // clear hull list
+fClearCollisions() : int                                  // clear collisions
+fDetectCollisions() : int                                 // detect collisions
+fResolveCollisions() : int                                // resolve detected

+sfCollisionTest(collision_out : cCollision3D,
    hullA : cCollisionHull3D, hullB : cCollisionHull3D) : bool
+sfCollisionTest_..._...(collision_out : cCollision3D,
    hullA : cCollisionHull...3D, hullB : cCollisionHull...3D) : bool
// test two hulls of specific types (ellipsis replaced with hull type)
// example tests: _SPHERE_SPHERE; _SPHERE_AABB/_AABB_SPHERE;
//  _SPHERE_BOX/_BOX_SPHERE; _AABB_AABB; _AABB_BOX/_BOX_AABB; _BOX_BOX;
-------------------------------------------------------------------------------
```