# Isopod Collision and Physics System: Technical Design Document

## Overview

This document describes the physics system to be updated for the next Isopod Labs game.

## Goals

These are some of the goals of the new system:

### Physics:

* Ability to simulate large (thousands) of objects at good frame rates.
* Integration of physics and collision so that things will collide and react automatically.
* Take advantage of the multi-processing ability of consoles.
* Stable resting states, so that dropped boxes, for instance, come to rest naturally.
* Automatic optimization of non-moving items, so stationary, un-pushed items are completely taken out of the simulation loop, and added back in as soon as a non-resting force acts on them.
* Complex systems of constraints between bodies
* Cloth and hair simulation.
* Ragdoll physics
* Blending of physics information into the animation system

### Collision:

* Convex hull support
* Complex, compound collision.
* Collision Hierarchies for early rejection.
* Collision masks to prevent certain collision primitives interacting.
* Continuous collision detection – to prevent tunneling, and the need to divide time into small pieces globally.
* Support for non-uniformly scaled objects

### Wish list:

* Global support for patch collision
* Soft-body dynamics
* Fluid simulation – including smoke

## Pipeline

Collision primitives will be authored in 3DS Max. Because we want the procedural information, we use the mesh primitives such as Sphere, Cylinder, Capsule, etc. To identify these as collision primitives, the node name must begin with ‘COLL\_’. The benefit of this technique is the ability to see the primitives in 3DS Max, and to be able to readily identify them as collision information.

Alternatively, primitives can be specified as entities using the IsoEntity modifier. These primitives would usually be generated using scripted ix support from within Max. A benefit of this technique is clutter is avoided.

A script can generate the smallest enclosing Sphere, Box (aligned or unaligned to the axes), Capsule and Cylinder as either a Max primitive, or as an entity. We will add scripts to enclose concave objects with multiple convex hulls, with adjustable complexity, and add support for automatic generation of collision hierarchies for complex shapes.

Physical information about the simulated object is encoded in a ‘MassProperties’ entity, which specifies mass, moments of inertia and centre of mass. These are given in units that correspond to a density of water. The actual density is specified separately, and is used to scale the mass and inertia values.

A physically-simulated object can be created by using the entity ‘PhysicsObject’, which specifies material properties, such as density, friction and restitution. It probably makes sense to move this information into the collision data, through an id that is used as a material lookup. Multiple collision primitives on the same object could then have separate ids – as could separate facets of a hull collision.

Whether the collision is defined using primitives or entities, the data is exported using the ix system as entity data attached to exported node. This data needs no further modification, since it is not platform-specific or bulky enough to require compression.

## Runtime Component

As a node is instantiated, the collision entities are used to generate runtime collision information. The entity data is not used directly at runtime in order to keep all the collision information in a cache-friendly format, and with additional runtime data for optimization. Similarly, rigid bodies are created from the MassProperties.

### Collision

Each root-level collision will maintain an axis-aligned bounding box, which should take the movement of the associated rigid body into account, for continuous collision.

A secondary bounding box shall be maintained that would enclose the collision assuming the associated rigid body continued its current motion for some (TBD) number of frames. This box will then only be updated when the collision no longer fits inside it.

The collision will be inserted into a spacially divided hierarchy based on this larger box. The position in the tree will then only need to be updated when this box has to change. In this case, the collision will be removed from the tree, and re-inserted.

Non-moving objects can set both boxes to the bounds of collision primitives, and will never need to be moved within the tree. Thus non-moving objects incur no runtime cost apart from the fact that their presence makes the tree bigger, and slower to traverse.

This tree will be used for broad-phase collision. Overlapping boxes can be instantly detected from the arrangement of the tree, and then the smaller boxes inside the nodes can be checked.

All groupings of collisions must then be saved in separate lists, and time subdivision is applied. As collisions are determined to be invalid after time subdivision, the lists can be further split in order to avoid subdividing unnecessarily.

Ultimately, true collisions can then calculate the area and angle of contact, and these will be added into the physics simulation as temporary constraints.

### Physics

The positions and orientations of the rigid bodies are updated via the angular and linear velocities before the collision phase. After the collisions have been determined, groupings of inter-constrained bodies are then grouped together into simulation ‘islands’.

The constraints are evaluated and forces are applied to counter the discrepancy between the actual and desired positions, orientations or velocities. If there is more than one constraint on a body, then multiple iterations of this simulation may need to be carried out until the constraints are within a given tolerance.

It is also possible to build the adjustments into a large linear system to be evaluated together. This may reduce the number of iterations needed.