Rigid Body Physics in USD Proposal

Purpose and Scope

With the rising adoption of USD across domains, it is being applied in a much broader set of applications than originally envisioned. One such application is the authoring, interchange and delivery of interactive 3D graphics content. Examples for such content in the consumer space include computer games and 3D web applications. Such applications often include real-time physics simulations to allow realistic user interaction with virtual objects. In professional and academic applications, there are a number of use cases in e.g. mechanical engineering, architecture, artificial intelligence and robotics where vehicles or robots are designed, tested and trained in simulation. Our schema proposal is to extend USD to represent the data needed by such simulation applications.

The space of all types of simulation is enormous. We see this proposal as version one of a sequence of extensions that start with the most basic and common concepts, and we intend to incrementally add more capabilities in the future. This proposal focuses specifically on rigid body physics.

Overall Design Concerns

This first proposal will concern only rigid body simulations. Rigid body simulations are the most broadly applicable category we could identify, with common and long standing uses across all disciplines described above.

Rigid Body Simulation Primer

Fundamentally, rigid body simulators take as input a list of rigid bodies and a list of constraints. Given the state of the bodies at the current time, they compute the updated state of the bodies a moment in time later, with the general desire being that the bodies' movement while constrained by the constraints obeys the laws of physics. One can invoke a sequence of such simulation updates to generate an animation.

A rigid body can be described by its pose (position and orientation in a well defined frame of reference), as well as its mass distribution (specified by a center of mass position, total mass, and an inertia tensor). The body will also have a velocity (linear and angular vectors). Pose and velocity are both inputs and outputs of the simulation update.

Constraints can take many forms, but fall primarily into two categories:

- Explicit constraints, often called joints, which create a fixed relationship between two rigid bodies. One example is a requirement that one body never rotate relative to the other body, even if relative translation is possible.
- Implicit constraints, most commonly contacts, which are generally created 'behind the scenes' by the simulator to ensure that e.g. solid objects do not pass through each other. For

the simulator to derive these constraints, each body must be provided with a collision representation and physical material properties.

Simulations often share a set of global parameters that influence the simulation of all bodies. It is generally possible to simultaneously create multiple simulations, each with their own set of parameter settings.

USD Implementation

Disambiguation

First, it is clear that some terminology commonly used by the physics simulation community, such as 'scene', 'joint', and 'material' have different meanings than in VFX and as already used by USD, so we decided to prefix all of our schema classes with 'Physics' and also make use of namespacing to avoid any ambiguity.

Fundamental Editing Capabilities

A primary assumption in designing this schema was that one of the most common use cases will be to add physics behavior to existing USD content. Furthermore, the conventional wisdom was that to maximize the performance of USD implementations, it is best to avoid inflating the number of USD objects in a scene. Thus we believe the best approach is to attach new API schemas that contain physics attributes to existing USD objects whenever this makes sense. In rare cases there is no object already available to which simulation attributes can be attached in a rational manner, and in these cases we decided to create new USD IsA schemas.

It is vital that any operation to add physics can also be undone; we will be leveraging the recently added RemoveAPI() capability.

Similarly, in editor use cases it is a common capability to temporarily be able to mute/disable properties, without deleting them outright. Deletion has the disadvantage that the stored settings are lost completely. USD allows entire objects to deactivate via an active flag, but this is not possible per-API. In a few cases muting behavior is a really common use case. For these cases we have defined a boolean enable attribute. (Note that we initially wanted to have the enable flag in a base class for the classes that need it, but this creates problems when multiple enableable APIs are applied to an object. In this case USD only creates a single shared enable flag, which is not what we want.)

Physics Scenes

As discussed above, we wish to enable multiple independent physics simulations within a single USD stage. We found the best way to do this is to create a PhysicsScene class. It was proposed to use the USD layers concept to partition physics into separate scenes, but we were concerned that the stage concept is already so overused for many different things (e.g. collaboration, version control) that we would prefer to avoid stretching it to yet another use case. In case there are multiple scenes in a stage, bodies are assigned to specific scenes using a rel from body to scene. If there is only one unique scene, an explicit rel is unnecessary, and bodies are assumed to be associated with the singleton scene. It is not possible to put a single body into multiple scenes as they would all be trying to influence and write conflicting information into such a body.

Scenes can define a gravity vector attribute which accelerates all contained bodies appropriately. Gravity is provided as a separate direction vector and magnitude. This is so that a default direction (negative stage up axis) and a default magnitude (earth gravity) can be requested independently.

Types

USD differentiates between base and role value types. We tried to use the available role types whenever applicable. For example, a velocity is a vector3f rather than a float3.

We chose to use single rather than double precision floats as widely available real time physics simulation software universally use single precision types for best performance, and the use of double or extended precision is only warranted for positions in extremely large spaces, which is already covered by making use of USD's built-in xform type.

Units

In terms of units, physics makes use of USD's established concepts of distance and time, and also adds the concept of mass. All of the physical quantities we use can be decomposed into a product of these three basic types. USD does not prescribe units for distance and time. It however has the concept of metersPerUnit and timeCodesPerSecond metadata which makes it possible to scale content authored at different scales correctly relative to each other when bringing them into a shared scene. This physics extension respects this distance and time capability with physics, and adds a kilogramsPerUnit metadata which remains consistent with the SI system.

All one dimensional angular values are specified in degrees for reasons of content creator intuition and consistency with existing degree values in USD like camera FOV or Euler rotations.

In the schema we indicate the units for each specified quantity as an expression using the terms 'distance', 'degrees', 'mass' and 'time' as defined above. A USD stage can be composed by referencing a number of USD files each using their own distinct unit conversion metadata. This means that before simulation, all values can be converted using the respective unit conversion metadata into an implementation dependent common system of units before they can be simulated. Similarly, any simulation outputs can be converted back into their original units before being written back to USD.

Default Values

Some problems came up while specifying this schema in connection with default values. First, there is a recent change to USD that eliminates the possibility of not creating attributes for schema APIs, which used to be a convenient way to denote a request to use a default value for the attribute. We now instead specify default values explicitly, typically sentinel values that lie outside of the range of legal values for a particular attribute. For example, if an attribute is normally required to be non-negative, we use -1.0 to request a certain default behavior. Sometimes the attribute can use the entire floating point range, in which case we reserve what is effectively +/- infinity at the edges of this range as sentinels. We will use the floating point 'inf' literal which USD supports in files and schemas to denote this. We document such default sentinel behavior on a case by case basis in the schema.

Rigid Bodies

We represent physics rigid bodies using the PhysicsRigidBodyAPI, which can be applied to any UsdGeomXformable. UsdGeomXformable is the suitable base class as it provides a placement in space via the xform which is also a fundamental property of physics bodies.

Rigid bodies have linear and angular velocity attributes that are specified in local space, to be consistent with velocities in point instancers and a node's xform.

Bodies can specify a simulationOwner scene rel for the aforementioned multi-scene simulation scenario.

Interaction with the USD hierarchy

If a node in a USD scene graph hierarchy is marked with PhysicsRigidBodyAPI, the behavior is such that all children of the marked node are assumed to be part of this rigid body, and move rigidly along with the body. This is consistent with the common behavior one expects during hand-animation of a sub-tree. If aggregate properties of the entire rigid body must be computed, such as total mass or the entirety of its collision volume, then the contents of the entire subtree are considered.

Note that it is of course permitted to change/animate the transforms in such a sub-tree, in which case any derived quantities in the physics engine such as center of mass or relative shape poses will be updated. Such animation will however not generate momentum. For example, rapidly animating rigid portions of Luxo Jr. will not cause the lamp to jump, since to compute such behavior we would need to capture the relative masses of multiple independent portions of the lamp, which is not possible if the whole is treated as a single rigid assembly. The correct approach would be to model each of the rigid portions of the lamp as independent rigid bodies, and connect these with joints, which we will discuss later.

It is not possible to have nested bodies. PhysicsRigidBodyAPIs applied to anything in the subtree under a node that already has a PhysicsRigidBodyAPI are ignored.

Sleep

To make large terrestrial simulations possible where, generally, all bodies eventually fall to the ground and come to rest, most rigid body simulation software have the concept of 'sleeping' these bodies to improve performance. This means that interactions cease to be updated when an equilibrium state is reached, and start to be updated again once the equilibrium state has somehow been disturbed. It is also possible to start off simulations in a sleeping state. We provide PhysicsRigidBodyAPI:startsAsleep to support this. We have considered exposing the runtime sleep state of each body in the simulation so that it would be visible to USD when the simulation deactivated a body, and to let USD force a body to sleep during simulation. We decided against this since the precise deactivation rules are an implementation detail that can vary significantly between simulations, so we prefer to keep this as a hidden implementation detail for the time being.

Kinematic Bodies

In games and VFX it is often desirable to have an animator take full control over a body, even as it interacts with other physics driven bodies. We call such bodies 'kinematic'. Kinematic bodies still 'pull on' joints and 'push on' touching rigid bodies, but their xform is only read, but not written, by the physics simulator, letting the animation system write their xforms. We support such bodies using

the PhysicsRigidBodyAPI:kinematicEnabled attribute. Kinematic bodies are not exactly the same thing as an animated static body with a collider: The simulation infers a continuous velocity for the kinematic body from the keyframing, and this velocity will be imparted to dynamic bodies during collisions.

Animation of Attributes

We worked with the assumption that every attribute on every class that is not explicitly marked with "uniform" can be animated. Obviously erratic changing of some parameters could make some simulations explode in practice, but we believe this is highly implementation dependent and not a reason to generally forbid attribute animation.

Body Mass Properties

We opted to decouple mass properties from PhysicsRigidBodyAPI and place them in a separate PhysicsMassAPI. PhysicsMassAPI is not required in most common cases where the mass properties of an object can be derived from collision geometry (discussed further down in this document) and the PhysicsMaterialAPI. Most commonly, PhysicsMassAPI is applied in addition to PhysicsRigidBodyAPI.

Unlike PhysicsRigidBodyAPI, it is also possible to apply PhysicsMassAPI multiple times in a USD scene graph subtree, in order to make it possible to accumulate the mass of rigid components.

The mass of an object may be specified in multiple ways, and several conflicting settings are resolved using a precedence system that will initially seem rather complex yet but is actually intuitive and practical:

- Parents' explicit total masses override any mass properties specified further down in the subtree.
- Density has lower precedence than mass, so explicit mass always overrides implicit mass that can be computed from volume and density.
- A density in a child overrides a density specified in a parent for all of the subtree under the child.
- A density specified via PhysicsMassAPI, even if it is inherited from a node higher in the tree, overrides any density specified via a material (see PhysicsMaterialAPI later in this document).
- Implicit mass at any node is the computed volume of collision geometry at that node times the locally effective density, plus the implicit masses of all children in the subtree.
- Density is assumed to be 1000.0 kg/m^3 (approximately the density of water) for volume computation when no other density is specified locally, or in rel-ed materials either locally or higher up in the tree, and this value is converted into the collider's native units prior to being used for mass computation.
- Mass is assumed to be 1.0 in the mass units used when none is provided explicitly, and there are no collision volumes to derive from.

Since implementing this rule set is maybe nontrivial, we plan to make the pseudocode of a mass computation system available that relies on the underlying physics system to compute the volume of collision geometry.

Collision Shapes

Our design for collision shapes defines a PhysicsCollisionAPI that may be attached to objects of type USDGeomGprim representing graphics geometry. Specifically, we suggest the support of USDGeomCapsule, USDGeomCone, USDGeomCube, USDGeomCylinder, USDGeomSphere and USDGeomMesh, though the precise set of supported geoms might be implementation specific. Note also that some implementations might support some of these shapes using potentially faceted convex approximations.

As we have perhaps already alluded to, a subtree under a PhysicsRigidBodyAPI node may contain multiple collision shape nodes (or 'colliders') that are required to resolve the motion of the body as it touches other bodies. For example, a teapot is a single rigid body (the top level node is marked with PhysicsRigidBodyAPI), but it may be composed of multiple Mesh and other Geoms at and under this node. Each of these parts can gain a PhysicsCollisionAPI which instructs the system to make this shape's geom into a collider for the purposes of physics simulation.

It is also possible to have PhysicsCollisionAPIs on nodes that are not under a PhysicsRigidBodyAPI. These are treated as static colliders -- shapes that are not moved by physics, but they can still collide with bodies, at which point they are interpreted as having zero velocity and infinite mass.

Note that for this static collider case when we do not have a relevant PhysicsRigidBodyAPI, it is possible for the PhysicsCollisionAPI to specify a simulationOwner scene. If there is a PhysicsRigidBodyAPI that this collider belongs to, the collider's simulation owner attribute is ignored.

Note that since according to USD rules, USDGeomGprims must be generally be leaf nodes, and because PhysicsCollisionAPI can only be applied to USDGeomGprim, it means that there is no opportunity to inherit PhysicsCollisionAPI attributes down the scene graph. If a mesh is composed of submeshes, all of the submeshes are considered to be part of the collider.

It is worth pointing out that this present design does have the drawback that it is not possible to add multiple colliders to a single geom object directly. To add multiple colliders one must create a parent Xform (which receives the PhysicsRigidBodyAPI), and then add the original geom as a child, and add any additional colliders as additional children. This is a bit more invasive than we would prefer, but the only alternative would be to make colliders Is-A schemas rather than APIs, which there was a desire to avoid to prevent the number of USD objects from increasing a great deal.

Turning Meshes into Shapes

Simple USD Prims like Sphere, Cylinder, Cube, Cone and Capsule are generally able to be used for physics simulation directly with the simple addition of a PhysicsCollisionAPI. USDMesh is a bit tricky because the state of the art in simulating arbitrary meshes in real time comes with some tradeoffs that users generally want control over. To support this, we allow PhysicsMeshCollisionAPI to be applied to USDGeomMeshes only, alongside the PhysicsCollisionAPI. This API has an approximation attribute that lets the user choose between no approximation (generally lowest performance), a simplified mesh, a set of convex hulls, a single convex hull, a bounding box or a bounding sphere. If an implementation does not support a particular kind of approximation, it is recommended that it falls back to the most similar supported option.

One may specify a collision mesh explicitly (for example one that was processed by a particular decimator) by adding the custom collider mesh as a sibling to the original graphics mesh, set it to

'guide' so it does not render, and apply PhysicsCollisionAPI and PhysicsMeshCollisionAPI to it specifying no approximation.

Physics Materials

Just like graphics, physics uses material properties. These are primarily used to inform friction and collision restitution behavior, in addition to being one of several ways to specify object density as discussed earlier. All these properties are stored in the PhysicsMaterialAPI, which can be applied to a USD Material node as we believe it to be practical to add physics properties to an established USD material library.

PhysicsMaterials are bound in the same way as graphics materials using material:binding, either with no purpose qualifier or with a specific 'physics' purpose. Note that this approach also permits using binding different materials to GeomSubsets. Not all physics simulations support different materials per GeomSubset, and it's possible that all but one subset per collider will be ignored by the implementation.

The unitless material coefficients dynamicFriction, staticFriction, and restitution should be familiar to anyone who knows high school physics.

Plane Shapes

Implicit plane shapes are a very common physics primitive used primarily for testing simple simulations. There are plans to add a Plane class to USD as a USDGeomGPrim. We look forward to supporting such plane shapes as static colliders when they become available.

Collision Filtering

Even in the simplest practical applications, the need to ignore some collisions occurs often. One might need the sword of a game character to pass through an enemy rather than to bounce off, while wanting it to bounce off walls, for example.

We define a CollisionGroup as an IsA schema with a UsdCollectionAPI applied, that defines the membership of colliders (objects with a PhysicsCollisionAPI) in the group. Each group also has a list of rel-s to other groups (potentially including itself) with which it needs to not collide. Colliders not in any CollisionGroup collide with all other colliders in the scene.

Pairwise Filtering

Sometimes group based filtering is insufficiently powerful to take care of some filtering special cases. One would for example set up group based filtering such that bodies of human characters collide against extremities like arms and legs, generally assuming that these arms and legs belong to different humans than the bodies. One however often doesn't want the extremities of a particular human to collide with its own body, which is hard to avoid during a lot of constant close proximity movement. To cover this case we have the FilteringPairsAPI, which holds a list of relationships to other objects with which collisions are explicitly disabled. This pairwise filtering has precedence over group based filtering.

The FilteringPairsAPI can be applied to objects with a PhysicsRigidBodyAPI, PhysicsCollisionAPI, or PhysicsArticulationAPI.

It is sufficient to have a rel from an object A to an object B, to get the filtering behavior. In this case the backwards rel from B to A is implicit and not necessary.

Joints

Joints are generally fixed attachments that can represent the way a drawer is attached to a cabinet, a wheel to a car, or links of a robot to each-other. Here we try to focus on a set of capabilities that are common to most simulation packages and sufficiently expressive for a large number of applications.

Mathematically, jointed assemblies can be modeled either in maximal (world space) or reduced (relative to other bodies) coordinates. Both representations have pros and cons. We are proposing a USD representation that will work with both approaches.

Joint Reference Frames

Our joint base type is the IsA class PhysicsJoint. Joints don't necessarily have a single unique Xform in space, rather, they are defined by two distinct frames, one relative to each of the two bodies which they connect.

These two frames might not work out to be the same position and orientation in world space because of either the permitted relative movement of the joint (think of a car suspension moving up and down: the joint frame of the suspension is constant relative to both the car body and the car axle, yet the axle and undercarriage move relative to each other) or the error of approximate simulations that can permit the joint to slightly pull apart when subjected to significant forces or velocities.

Because of these dual transforms, it did not make sense for us to derive PhysicsJoint from Xformable, which just has one Xform. We could have created an asymmetrical solution where the secondary xform is added on, or split the joint object into two separate joint frames that are parented into the scene graph and are then somehow pairwise cross referenced, but we opted to go with an entirely new class that has all the information we need in a symmetrical fashion.

Jointed Bodies

A joint rels one or two rigid bodies (which must have a PhysicsRigidBodyAPI). One body rel can be null in which case the first body is jointed to the world frame. Henceforth whenever we mention the two bodies, it is assumed that one of them may implicitly be the static world coordinate frame.

Note that we currently do not require support for two plausible scenarios: One, to create a joint to a scene graph node with no PhysicsRigidBodyAPI either locally or in any parent. We could plausibly treat such a node as another way to joint to the world frame, only with an additional relative transform. Second, to create a joint to direct or indirect children of a node with a PhysicsRigidBodyAPI. This would be equivalent to jointing to the rigid body, only with an additional relative transform.

The joint space relative to each body is a translation and orientation only, scaling is not supported (This is a general tension between graphics and physics. In the real world it is generally not possible to scale real objects and simulations do not tend to support scaling during rigid body simulation). For this reason we don't use a general USD xform that is too flexible for our needs, but rather a separate position and orientation quaternion. (Note however that this local joint space is fixed in the node's local space, which of course CAN be scaled using the node's own Xform scaling. This means

that if a doorknob is attached to a door at a particular position, it will continue to appear in the same correct position on the door regardless of how the door is scaled, without having to adjust the joint position.)

Joint Collision Filtering

It is common practice to disable collisions between jointed objects so that their collision shapes don't interfere, and this is therefore the default behavior that can be changed using the joint's collisionEnabled attribute. This only applies to joints with two explicit bodies: a joint to the world does NOT disable collisions between the body and the world.

Breaking and Disabling Joints

One property we believe can be practical for all joints is that they can break when sufficient force is applied. For example a door can be ripped off its hinges. This can be modeled using the breakForce and breakTorque attributes.

Joints can entirely be temporarily disabled just like rigid bodies or colliders. Contrary to breaking, which is a (within a simulation run irreversible) simulated behavior, disabling is a request to not simulate the joint at all.

Joint Subtypes

Joints have a number of possible derived types that allow for specific types of joints, however, it can also be used to represent a generic configurable joint, so in that sense it is not an abstract type.

The subtypes PhysicsSphericalJoint, PhysicsRevoluteJoint and PhysicsPrismaticJoint both define a primary axis (Following the USD axis definition pattern established in e.g. GeomCapsule and GeomCylinder) and a top and bottom motion limit along it.

PhysicsDistanceJoint defines a min and max distance between the attachment points. The PhysicsFixedJoint has no additional properties and simply locks all relative degrees of freedom.

Joint Limits and Drives

Instead of using one of the predefined joint subtypes, it is also possible to compose a custom joint from a set of limits and drives. Limits and drives are multi-apply schemas, so one can apply multiple instances, one for each degree of freedom. The degree of freedom is specified via the TfToken (effectively a string, one of "transX", "transY", "transZ", "rotX", "rotY", "rotZ", "distance", that is postpended after the class name.)

The limit API further contains optional low and high limit attributes.

The drive API allows joints to be motorized along degrees of freedom. It may specify either a force or acceleration drive (The strength of force drives is impacted by the mass of the bodies attached to the joint, an acceleration drive is not). It also has a target value to reach, and one can specify if the target is a goal position or velocity. One can limit the maximum force the drive can apply, and one can specify a spring and damping coefficient.
The resulting drive force or acceleration is proportional to

$$stiffness \times (targetPosition - p) + damping \times (targetVelocity - v)$$

where p is the relative pose space motion of the joint (the axial rotation of a revolute joint, or axial translation for a prismatic joint) and v is the rate of change of this motion.

For all limits that specify ranges, a "low" limit larger than the "high" limit means the joint motion along that axis is locked.

Articulations

Above we did say that we also support reduced coordinate joints, which require some additional specification. We decided to do this with a minimal extension of the above maximal joints. Any node of the USD scene graph hierarchy may be marked with an ArticulationRootAPI. This informs the simulation that any joints found in the subtree should preferentially be simulated using a reduced coordinate approach. For floating articulations (robotics jargon for something not bolted down, e.g. a wheeled robot or a quadcopter), this API should be used on the root body (typically the central mass the wheels or rotors are attached to), or a direct or indirect parent node. For fixed articulations (robotics jargon for e.g. a robot arm for welding that is bolted to the floor), this API can be on a direct or indirect parent of the root joint which is connected to the world, or on the joint itself. If there are multiple qualifying bodies or joints under an ArticulationRootAPI node, each is made into a separate articulation root.

This should in general make it possible to uniquely identify a distinguished root body or root joint for the articulation. From this root, a tree of bodies and joints is identified that is not to contain loops (which may be closed by joint collections). If loops are found, they may be broken at an arbitrary location. Alternatively, a joint in the loop may use its excludeFromArticulation attribute flag to denote that it wishes to remain a maximal joint, and at this point the loop is then broken.

Concrete Schemas

Here is our concrete schema proposal in full:

```
#usda 1.0
(
        subLayers = [
    @usdGeom/schema.usda@
        ]
        kilogramsPerUnit = 1.0
)

over "GLOBAL" (
        customData = {
        string libraryName = "physicsSchema"
        string libraryPath = "physicsSchema"
        string libraryPrefix = "PhysicsSchema"
        }
)
{

}

class PhysicsScene "PhysicsScene"
(
    customData = {
        string className = "PhysicsScene"
        }
    doc = """General physics simulation properties, required for simulation."""

    inherits = </Typed>
)
```

```
{
    vector3f physics:gravityDirection = (0.0, 0.0, 0.0) (
        customData = {
            string apiName = "gravityDirection"
        }
        doc = """Gravity direction vector in simulation word space. Will be
        normalized before use. A zero vector is a request to use the negative
        upAxis. Unitless."""
    )

    float physics:gravityMagnitude = -inf (
        customData = {
            string apiName = "gravityMagnitude"
        }
        doc = """Gravity acceleration magnitude in simulation word space.
        A negative value is a request to use a value equivalent to earth
        gravity regardless of the metersPerUnit scaling used by this scene.
        Units: distance/time/time."""
    )
}

class "PhysicsRigidBodyAPI"
(
        customData = {
            string className = "PhysicsRigidBodyAPI"
        }
    doc = """Applies physics body attributes to any UsdGeomXformable prim and
    marks that prim to be driven by a simulation. If a simulation is running
    it will update this prim's pose. All prims in the hierarchy below this
    prim should move accordingly."""

    inherits = </APISchemaBase>
)
{

    bool physics:rigidBodyEnabled = true (
        customData = {
            string apiName = "rigidBodyEnabled"
        }
        doc = """Determines if this PhysicsRigidBodyAPI is enabled."""
    )

       bool physics:kinematicEnabled = false (
        customData = {
            string apiName = "kinematicEnabled"
        }
        doc = """Determines whether the body is kinematic or not. A kinematic
        body is a body that is moved through animated poses or through
        user defined poses. The simulation derives velocities for the
        kinematic body based on the external motion.  When a continuous motion
        is not desired, this kinematic flag should be set to false."""
    )

    rel physics:simulationOwner (
        customData = {
            string apiName = "simulationOwner"
        }
        doc = """Single PhysicsScene that will simulate this body.  By
        default this is the first PhysicsScene found in the stage using
        UsdStage::Traverse()."""

    )

       uniform bool physics:startsAsleep = false (
        customData = {
            string apiName = "startsAsleep"
        }
        doc = "Determines if the body is asleep when the simulation starts."
    )

    vector3f physics:velocity= (0.0, 0.0, 0.0) (
```

```
        customData = {
            string apiName = "velocity"
        }
        doc = """Linear velocity in the same space as the node's xform.
        Units: distance/time."""
    )

    vector3f physics:angularVelocity = (0.0, 0.0, 0.0) (
        customData = {
            string apiName = "angularVelocity"
        }
        doc = """Angular velocity in the same space as the node's xform.
        Units: degrees/time."""
    )


}

class "PhysicsMassAPI"
(
        customData = {
            string className = "PhysicsMassAPI"
        }
    doc = """Defines explicit mass properties (mass, density, inertia etc.).
    MassAPI can be applied to any object that has a PhysicsCollisionAPI or
    a PhysicsRigidBodyAPI."""

    inherits = </APISchemaBase>
)
{
        float physics:mass = 0.0 (
        customData = {
            string apiName = "mass"
        }
        doc = """If non-zero, directly specifies the mass of the object.
        Note that any child prim can also have a mass when they apply massAPI.
        In this case, the precedence rule is 'parent mass overrides the
        child's'. This may come as counter-intuitive, but mass is a computed
        quantity and in general not accumulative. For example, if a parent
        has mass of 10, and one of two children has mass of 20, allowing
        child's mass to override its parent results in a mass of -10 for the
        other child. Note if mass is 0.0 it is ignored. Units: mass.
        """
    )

        float physics:density = 0.0 (
        customData = {
            string apiName = "density"
        }
        doc = """If non-zero, specifies the density of the object.
        In the context of rigid body physics, density indirectly results in
        setting mass via (mass = density x volume of the object). How the
        volume is computed is up to implementation of the physics system.
        It is generally computed from the collision approximation rather than
        the graphical mesh. In the case where both density and mass are
        specified for the same object, mass has precedence over density.
        Unlike mass, child's prim's density overrides parent prim's density
        as it is accumulative. Note that density of a collisionAPI can be also
        alternatively set through a PhysicsMaterialAPI. The material density
        has the weakest precedence in density definition. Note if density is
        0.0 it is ignored. Units: mass/distance/distance/distance."""
    )

        point3f physics:centerOfMass = (0.0, 0.0, 0.0) (
        customData = {
            string apiName = "centerOfMass"
        }
        doc = """Center of mass in the prim's local space. Units: distance."""
    )

        float3 physics:diagonalInertia = (0.0, 0.0, 0.0) (
```

```
        customData = {
            string apiName = "diagonalInertia"
        }
        doc = """If non-zero, specifies diagonalized inertia tensor along the
        principal axes. Note if diagonalInertial is (0.0, 0.0, 0.0) it is
        ignored. Units: mass*distance*distance."""
    )

        quatf physics:principalAxes = (1, 0, 0, 0) (
        customData = {
            string apiName = "principalAxes"
        }
        doc = """Orientation of the inertia tensor's principal axes in the
        prim's local space."""
    )
}

class "PhysicsCollisionAPI"
(
        customData = {
            string className = "PhysicsCollisionAPI"
        }
    doc = """Applies collision attributes to a UsdGeomXformable prim. If a
    simulation is running this geometry is colliding with other geometries that
    do have PhysicsCollisionAPI applied. If a prim in the parent hierarchy does have
    the PhysicsAPI applied, this collision is a part of that body. If there is
    no body in the parent hierarchy, this collision is considered to be a static
    collision."""

    inherits = </APISchemaBase>
)
{
    bool physics:collisionEnabled = true (
        customData = {
            string apiName = "collisionEnabled"
        }
        doc = """Determines if the PhysicsCollisionAPI is enabled."""
    )


    rel physics:simulationOwner (
        customData = {
            string apiName = "simulationOwner"
        }
        doc = """Single PhysicsScene that will simulate this collider.
        By default this object belongs to the first PhysicsScene.
        Note that if a PhysicsAPI in the hierarchy above has a different
        simulationOwner then it has a precedence over this relationship."""
    )
}

class "PhysicsMeshCollisionAPI"
(
        customData = {
            string className = "PhysicsMeshCollisionAPI"
        }
       doc = """Attributes to control how a Mesh is made into a collider.
       Can be applied to only a USDGeomMesh in addition to its
       PhysicsCollisionMeshAPI."""

   inherits = </APISchemaBase>
)
{
        uniform token physics:approximation = "none" (
        customData = {
            string apiName = "approximation"
        }
        allowedTokens = ["none","convexDecomposition","convexHull",
                "boundingSphere","boundingCube","meshSimplification"]

        doc = """Determines the mesh's collision approximation:
```

```
        "none" - The mesh geometry is used directly as a collider without any
    approximation.
        "convexDecomposition" - A convex mesh decomposition is performed. This
    results in a set of convex mesh colliders.
        "convexHull" - A convex hull of the mesh is generated and used as the
    collider.
        "boundingSphere" - A bounding sphere is computed around the mesh and used
    as a collider.
        "boundingCube" - An optimally fitting box collider is computed around the
    mesh.
        "meshSimplification" - A mesh simplification step is performed, resulting
    in a simplified triangle mesh collider."""
        )
}

class "PhysicsMaterialAPI"
(
        customData = {
            string className = "PhysicsMaterialAPI"
        }
    doc = """ Defines simulation material properties. All collisions that
    have a relationship to this material will have their collision response
    defined through this material."""

    inherits = </APISchemaBase>
)
{
    float physics:dynamicFriction = 0.0 (
        customData = {
            string apiName = "dynamicFriction"
        }
        doc = """Dynamic friction coefficient. Unitless."""
    )

    float physics:staticFriction = 0.0 (
        customData = {
            string apiName = "staticFriction"
        }
        doc = """Static friction coefficient. Unitless."""
    )

    float physics:restitution = 0.0 (
        customData = {
            string apiName = "restitution"
        }
        doc = """Restitution coefficient. Unitless."""
    )

    float physics:density = 0.0 (
        customData = {
            string apiName = "density"
        }
        doc = """If non-zero, defines the density of the material. This can be
        used for body mass computation, see PhysicsMassAPI.
        Note that if the density is 0.0 it is ignored.
        Units: mass/distance/distance/distance."""
    )
}

class PhysicsCollisionGroup "PhysicsCollisionGroup"
(
        customData = {
            string className = "PhysicsCollisionGroup"
        }
    doc = """Defines a collision group for coarse filtering. When a collision
    occurs between two objects that have a PhysicsCollisionGroup assigned,
    they will collide with each other unless this PhysicsCollisionGroup pair
    is filtered. See filteredGroups attribute.

    A CollectionAPI:colliders maintains a list of PhysicsCollisionAPI rel-s that defines the
    members of this Collisiongroup.
```

```
    """

    inherits = </Typed>

 customData = {
        string extraIncludes = """
#include "pxr/usd/usd/collectionAPI.h" """
    }
prepend apiSchemas = ["CollectionAPI:colliders"]

)
{

        rel physics:filteredGroups (
        customData = {
            string apiName = "filteredGroups"
        }
        doc = """Rels  a list of PhysicsCollisionGroups with which
        collisions should be ignored."""
    )
}

class "PhysicsFilteredPairsAPI"
(
        customData = {
            string className = "PhysicsFilteredPairsAPI"
        }
    doc = """API to describe fine-grained filtering. If a collision between
    two objects occurs, this pair might be filtered if the pair is defined
    through this API. This API can be applied either to a body or collision
    or even articulation. The "filteredPairs" defines what objects it should
    not collide against. Note that FilteredPairsAPI filtering has precedence
    over CollisionGroup filtering."""

    inherits = </APISchemaBase>
)
{
    rel physics:filteredPairs (
        customData = {
            string apiName = "filteredPairs"
        }
        doc = """Relationship to objects that should be filtered."""
    )
}




class PhysicsJoint "PhysicsJoint"
(
        customData = {
            string className = "PhysicsJoint"
        }
    doc = """A joint constrains the movement of rigid bodies. Joint can be
    created between two rigid bodies or between one rigid body and world.
    By default joint primitive defines a D6 joint where all degrees of
    freedom are free. Three linear and three angular degrees of freedom.
    Note that default behavior is to disable collision between jointed bodies.
    """

    inherits = </Typed>
)
{
    rel physics:body0 (
        customData = {
            string apiName = "body0"
        }
        doc = """Relationship to first rigid body (or empty string if to
        world)."""
    )
```

```
rel physics:body1 (
    customData = {
        string apiName = "body1"
    }
    doc = """Relationship to second rigid body (or empty string if to
    world)."""
)

point3f physics:localPos0 = (0.0, 0.0, 0.0) (
    customData = {
        string apiName = "localPos0"
    }
    doc = """Relative position of the joint frame to body0's frame."""
)

quatf physics:localRot0 = (1.0, 0.0, 0.0, 0.0) (
    customData = {
        string apiName = "localRot0"
    }
    doc = """Relative orientation of the joint frame to body0's frame."""
)

point3f physics:localPos1 = (0.0, 0.0, 0.0) (
    customData = {
        string apiName = "localPos1"
    }
    doc = """Relative position of the joint frame to body1's frame."""
)

quatf physics:localRot1 = (1.0, 0.0, 0.0, 0.0) (
    customData = {
        string apiName = "localRot1"
    }
    doc = """Relative orientation of the joint frame to body1's frame."""
)

bool physics:jointEnabled = true (
    customData = {
        string apiName = "jointEnabled"
    }
    doc = """Determines if the joint is enabled."""
)

bool physics:collisionEnabled = false (
    customData = {
        string apiName = "collisionEnabled"
    }
    doc = """Determines if the jointed bodies should collide or not."""
)

uniform bool physics:excludeFromArticulation = false (
    customData = {
        string apiName = "excludeFromArticulation"
    }
    doc = """Determines if the joint can be included in an Articulation."""
)

float physics:breakForce = inf (
    customData = {
        string apiName = "breakForce"
    }
    doc = """Joint break force. If set, joint is to break when this force
    limit is reached. (Used for linear DOFs.)
    Units: mass * distance / time / time"""
)

float physics:breakTorque = inf (
    customData = {
        string apiName = "breakTorque"
    }
    doc = """Joint break torque. If set, joint is to break when this torque
```

```
        limit is reached. (Used for angular DOFs.)
        Units: mass * distance * distance / time / time"""
    )

}

class PhysicsRevoluteJoint "PhysicsRevoluteJoint"
(
        customData = {
            string className = "PhysicsRevoluteJoint"
        }
    doc = """Predefined revolute joint type (rotation along revolute joint
    axis is permitted.)"""

    inherits = </PhysicsJoint>
)
{
    uniform token physics:axis = "X" (
        customData = {
            string apiName = "axis"
        }
        allowedTokens = ["X", "Y", "Z"]
        doc = """Joint axis."""
    )

    float physics:lowerLimit = -inf (
        customData = {
            string apiName = "lowerLimit"
        }
        doc = """Lower limit.  Units: degrees. -inf means not limited in
        negative direction."""
    )

    float physics:upperLimit = inf (
        customData = {
            string apiName = "upperLimit"
        }
        doc = """Upper limit.  Units: degrees. inf means not limited in
        positive direction."""
    )
}

class PhysicsPrismaticJoint "PhysicsPrismaticJoint"
(
        customData = {
            string className = "PhysicsPrismaticJoint"
        }
    doc = """Predefined prismatic joint type (translation along prismatic
    joint axis is permitted.)"""

    inherits = </PhysicsJoint>
)
{
    uniform token physics:axis = "X" (
        customData = {
            string apiName = "axis"
        }
        allowedTokens = ["X", "Y", "Z"]
        doc = """Joint axis."""
    )

    float physics:lowerLimit = -inf (
        customData = {
            string apiName = "lowerLimit"
        }
        doc = """Lower limit. Units: distance. -inf means not limited in
        negative direction."""
    )

    float physics:upperLimit = inf (
        customData = {
```

```
            string apiName = "upperLimit"
        }
        doc = """Upper limit. Units: distance. inf means not limited in
        positive direction."""
    )
}

class PhysicsSphericalJoint "PhysicsSphericalJoint"
(
        customData = {
            string className = "PhysicsSphericalJoint"
        }
    doc = """Predefined spherical joint type (Removes linear degrees of
    freedom, cone limit may restrict the motion in a given range.)  It allows
    two limit values, which when equal create a circular, else an elliptic
    cone limit around the limit axis."""

    inherits = </PhysicsJoint>
)
{
    uniform token physics:axis = "X" (
        customData = {
            string apiName = "axis"
        }
        allowedTokens = ["X", "Y", "Z"]
        doc = """Cone limit axis."""
    )

    float physics:coneAngle0Limit = -1.0 (
        customData = {
            string apiName = "coneAngle0Limit"
        }
        doc = """Cone limit from the primary joint axis in the local0 frame
        toward the next axis. (Next axis of X is Y, and of Z is X.)  A
        negative value means not limited. Units: degrees."""
    )

    float physics:coneAngle1Limit = -1.0 (
        customData = {
            string apiName = "coneAngle1Limit"
        }
        doc = """Cone limit from the primary joint axis in the local0 frame
        toward the second to next axis.  A negative value means not limited.
        Units: degrees."""
    )
}


class PhysicsDistanceJoint "PhysicsDistanceJoint"
(
        customData = {
            string className = "PhysicsDistanceJoint"
        }
    doc = """Predefined distance joint type (Distance between rigid bodies
    may be limited to given minimum or maximum distance.)"""

    inherits = </PhysicsJoint>
)
{
    float physics:minDistance = -1.0 (
        customData = {
            string apiName = "minDistance"
        }
        doc = """Minimum distance. If attribute is negative, the joint is not
        limited. Units: distance."""
    )

    float physics:maxDistance = -1.0 (
        customData = {
            string apiName = "maxDistance"
        }
```

```
        doc = """Maximum distance. If attribute is negative, the joint is not
        limited. Units: distance."""
    )
}

class PhysicsFixedJoint "PhysicsFixedJoint"
(
        customData = {
            string className = "PhysicsFixedJoint"
        }
    doc = """Predefined fixed joint type (All degrees of freedom are
    removed.)"""

    inherits = </PhysicsJoint>
)
{
}

class "PhysicsLimitAPI"
(
        customData = {
            string className = "PhysicsLimitAPI"
        }
    doc = """The PhysicsLimitAPI can be applied to a PhysicsJoint and will
    restrict the movement along an axis. PhysicsLimitAPI is a multipleApply
    schema: The PhysicsJoint can be restricted along "transX", "transY",
    "transZ", "rotX", "rotY", "rotZ", "distance". Setting these as a
    multipleApply schema TfToken name will define the degree of freedom the
    PhysicsLimitAPI is applied to. Note that if the low limit is higher than
    the high limit, motion along this axis is considered locked."""

    inherits = </APISchemaBase>

    customData = {
        token apiSchemaType = "multipleApply"
        token propertyNamespacePrefix  = "limit"
    }
)
{
    float physics:low = -inf (
        customData = {
            string apiName = "low"
        }
        doc = """Lower limit. Units: degrees or distance depending on trans or
        rot axis applied to. -inf means not limited in negative direction."""
    )

    float physics:high = inf (
        customData = {
            string apiName = "high"
        }
        doc = """Upper limit.  Units: degrees or distance depending on trans or
        rot axis applied to. inf means not limited in positive direction."""
    )
}

class "PhysicsDriveAPI"
(
        customData = {
            string className = "PhysicsDriveAPI"
        }
    doc = """The PhysicsDriveAPI when applied to any joint primitive will drive
    the joint towards a given target.  The PhysicsDriveAPI is a multipleApply
    schema: drive can be set per axis "transX", "transY", "transZ", "rotX",
    "rotY", "rotZ". Setting these as a multipleApply schema TfToken name will
    define the degree of freedom the DriveAPI is applied to. Each drive is an
    implicit force-limited damped spring:
    Force or acceleration = stiffness * (targetPosition - position)
    + damping * (targetVelocity - velocity)"""

    inherits = </APISchemaBase>
```

```
    customData = {
        token apiSchemaType = "multipleApply"
        token propertyNamespacePrefix  = "drive"
    }
)
{
    uniform token physics:type = "force" (
        customData = {
            string apiName = "type"
        }
        allowedTokens = ["force", "acceleration"]
        doc = """Drive spring is for the acceleration at the joint (rather
        than the force)."""
    )

    float physics:maxForce = inf (
        customData = {
            string apiName = "maxForce"
        }
        doc = """Maximum force that can be applied to drive. Units:
                if linear drive:  mass*DIST_UNITS/time/time
                if angular drive: mass*DIST_UNITS*DIST_UNITS/time/time
                inf means not limited.  Must be non-negative.
        """
    )

    float physics:targetPosition = 0.0 (
        customData = {
            string apiName = "targetPosition"
        }
        doc = """Target value for position. Units:
        if linear drive: distance
        if angular drive: degrees."""
    )

    float physics:targetVelocity = 0.0 (
        customData = {
            string apiName = "targetVelocity"
        }
        doc = """Target value for velocity. Units:
        if linear drive:  distance/time
        if angular drive: degrees/time."""
    )

    float physics:damping = 0.0 (
        customData = {
            string apiName = "damping"
        }
        doc = """Damping of the drive. Unitless."""
    )

    float physics:stiffness = 0.0 (
        customData = {
            string apiName = "stiffness"
        }
        doc = """Stiffness of the drive. Unitless."""
    )
}

class "PhysicsArticulationRootAPI"
(
        customData = {
            string className = "PhysicsArticulationRootAPI"
        }
    doc = """PhysicsArticulationRootAPI can be applied to a scene graph node,
    and marks the subtree rooted here for inclusion in one or more reduced
    coordinate articulations. For floating articulations, this should be on
    the root body.  For fixed articulations, this can be somewhere above the
    root joint, which is connected to the world."""
```

```
    inherits = </APISchemaBase>
)
{
}
```

Examples

Box on Box

```
#usda 1.0

#Very basic example for using the Physics USD schema.

#Should show a box shaped rigid body that will fall on a flat

#static box when simulated.

(

    defaultPrim = "World"

    endTimeCode = 100

    metersPerUnit = 0.01

    startTimeCode = 0

    timeCodesPerSecond = 24

    upAxis = "Z"

    #new mass scaling

    kilogramsPerUnit = 1.0

)

def Xform "World"

{

    #Scene mandatory for simulation.  By If rigid bodies don't explicitly

    #specify a scene, they implicitly belong to this one.

    #by default, the scene will have earth gravity.

    def PhysicsScene "PhysicsScene"

    {

    }

    #This cube becomes a rigid body and a collider thanks to two applied

    #schemas.

    def Cube "BoxActor" (

        prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI"]

    )

    {

        #optional non zero starting velocities.

        #All other physics behavior is left at defaults.

        vector3f physics:velocity = (2, 1, 2)
```

```
        vector3f physics:angularVelocity = (1, 0, 0)
        color3f[] primvars:displayColor = [(0.2784314, 0.4117647, 1)]
        double size = 25
        double3 xformOp:translate = (0, 0, 500)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
    #This cube becomes a static ground box because it only has a collider
    #but no rigid body.
    def Cube "Ground" (
        prepend apiSchemas = ["PhysicsCollisionAPI"]
    )
    {
        color3f[] primvars:displayColor = [(0.5, 0.5, 0.5)]
        #scale the cube to be flat
        float3 xformOp:scale = (750, 750, 10)
        uniform token[] xformOpOrder = ["xformOp:scale"]
    }
    #just to make this scene render pretty, not relevant for physics
    def SphereLight "SphereLight"
    {
        float intensity = 30000
        float radius = 150
        double3 xformOp:translate = (650, 0, 1150)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
}
```

Box on Quad

```
#usda 1.0
#More advanced example showing mesh collisions
#and center of mass offset.  We expect to have the
#quad be represented as a convex mesh and have the
#cube come to rest on it, balanced on one of its corners.
(
    defaultPrim = "World"
    endTimeCode = 100
    metersPerUnit = 0.01
    startTimeCode = 0
```

```
    timeCodesPerSecond = 24
    upAxis = "Z"
    kilogramsPerUnit = 1
)
def Xform "World"
{
    def PhysicsScene "PhysicsScene"
    {
    }
    #Added a mass API so we can offset the center of mass.
    def Cube "BoxActor" (
        prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI", "PhysicsMassAPI"]
    )
    {
        #explicit mass
        float physics:mass = 10.0
        #offset center of mass so the cube settles on its corner
        point3f physics:centerOfMass = (40.0, 40.0, 40.0)
        color3f[] primvars:displayColor = [(0.2784314, 0.4117647, 1)]
        double size = 25
        double3 xformOp:translate = (0, 0, 500)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
    #A quad mesh that serves as the ground.
    def Mesh "Ground" (
        prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsMeshCollisionAPI"]
    )
    {
        #approximate with a convex hull, if we remove this, it will
        #be used directly
        uniform token physics:approximation = "convexHull"
        uniform bool doubleSided = 1
        int[] faceVertexCounts = [4]
        int[] faceVertexIndices = [0, 1, 2, 3]
        normal3f[] normals = [(0, 0, 1), (0, 0, 1), (0, 0, 1), (0, 0, 1)]
        point3f[] points = [(-1, 1, 0), (1, 1, 0), (1, -1, 0), (-1, -1, 0)]
        color3f[] primvars:displayColor = [(0.5, 0.5, 0.5)]
```

```
        texCoord2f[] primvars:st = [(0, 1), (1, 1), (1, 0), (0, 0)] (
            interpolation = "varying"
        )
        float3 xformOp:scale = (750, 750, 750)
        uniform token[] xformOpOrder = ["xformOp:scale"]
    }
    def SphereLight "SphereLight"
    {
        float intensity = 30000
        float radius = 150
        double3 xformOp:translate = (650, 0, 1150)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
}
```

Spheres with Materials

```
#usda 1.0
#Shows two spheres, one with high
#and one with low bounce, by using materials.
#Also uses a trimesh ground quad.
(
    defaultPrim = "World"
    endTimeCode = 100
    metersPerUnit = 0.01
    startTimeCode = 0
    timeCodesPerSecond = 24
    upAxis = "Z"

    #new mass scaling
    kilogramsPerUnit = 1.0

)
def Xform "World"
{
    def PhysicsScene "PhysicsScene"
    {
    }
```

```
def Sphere "RegularSphere" (
    prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI"]
)
{
    rel material:binding:physics = </World/Looks/RegularMaterial> (
        bindMaterialAs = "weakerThanDescendants"
    )

    color3f[] primvars:displayColor = [(0.2784314, 0.4117647, 1)]
    double3 xformOp:translate = (0, 0, 500)
    float3 xformOp:scale = (25, 25, 25)
    uniform token[] xformOpOrder = ["xformOp:translate", "xformOp:scale"]
}
def Sphere "BouncySphere" (
    prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI"]
)
{
    rel material:binding:physics = </World/Looks/BouncyMaterial> (
        bindMaterialAs = "weakerThanDescendants"
    )

    color3f[] primvars:displayColor = [(0.8784314, 0.2117647, 0.1)]
    double3 xformOp:translate = (300, 0, 500)
    float3 xformOp:scale = (25, 25, 25)
    uniform token[] xformOpOrder = ["xformOp:translate", "xformOp:scale"]
}
def Mesh "Ground" (
    prepend apiSchemas = ["PhysicsCollisionAPI"]
)
{
    int[] faceVertexCounts = [4]
    int[] faceVertexIndices = [3, 2, 1, 0]
    normal3f[] normals = [(0, 0, 1), (0, 0, 1), (0, 0, 1), (0, 0, 1)]
    point3f[] points = [(-1, 1, 0), (1, 1, 0), (1, -1, 0), (-1, -1, 0)]
    color3f[] primvars:displayColor = [(0.5, 0.5, 0.5)]
    texCoord2f[] primvars:st = [(0, 1), (1, 1), (1, 0), (0, 0)] (
        interpolation = "varying"
```

```
        )
        float3 xformOp:scale = (750, 750, 750)
        uniform token[] xformOpOrder = ["xformOp:scale"]
    }

    def Scope "Looks"
    {
        def Material "RegularMaterial" (
            prepend apiSchemas = ["PhysicsMaterialAPI"]
        )
        {
            double density = 10
            float restitution = 0.1
        }
        def Material "BouncyMaterial" (
            prepend apiSchemas = ["PhysicsMaterialAPI"]
        )
        {
            double density = 10
            float restitution = 0.8
        }
    }
    def SphereLight "SphereLight"
    {
        float intensity = 30000
        float radius = 150
        double3 xformOp:translate = (650, 0, 1150)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
}
```

Group Filtering

```
#usda 1.0
#Shows two boxes that collide with a ground box
#but do not collide with each other thanks to
#group based filtering.
(
```

```
    defaultPrim = "World"
    endTimeCode = 100
    metersPerUnit = 0.01
    startTimeCode = 0
    timeCodesPerSecond = 24
    upAxis = "Z"
    kilogramsPerUnit = 1.0
)
def Xform "World"
{
    def PhysicsScene "PhysicsScene"
    {
    }
    def Cube "Box1" (
        prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI"]
    )
    {
        color3f[] primvars:displayColor = [(0.2784314, 0.4117647, 1)]
        double size = 25
        double3 xformOp:translate = (0, 0, 50)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
    def Cube "Box2" (
        prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI"]
    )
    {
        color3f[] primvars:displayColor = [(0.2784314, 0.4117647, 1)]
        double size = 25
        double3 xformOp:translate = (0, 0, 100)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
    def PhysicsCollisionGroup "DynamicGroup" (
    prepend apiSchemas = ["CollectionAPI:colliders"]
    )
    {
        prepend rel collection:colliders:includes = [
            </World/Box1>,
```

```
            </World/Box2>
        ]
        prepend rel physics:filteredGroups = [
            </World/DynamicGroup>
        ]
    }
    def Cube "Ground" (
        prepend apiSchemas = ["PhysicsCollisionAPI"]
    )
    {
        color3f[] primvars:displayColor = [(0.5, 0.5, 0.5)]
        float3 xformOp:scale = (750, 750, 10)
        uniform token[] xformOpOrder = ["xformOp:scale"]
    }
    def SphereLight "SphereLight"
    {
        float intensity = 30000
        float radius = 150
        double3 xformOp:translate = (650, 0, 1150)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
}
```

Pair Filtering

```
#usda 1.0
(
    defaultPrim = "World"
    endTimeCode = 100
    metersPerUnit = 0.01
    startTimeCode = 0
    timeCodesPerSecond = 24
    upAxis = "Z"
    kilogramsPerUnit = 1.0
)
def Xform "World"
{
    def PhysicsScene "PhysicsScene"
    {
```

```
    }
    def Cube "Box1" (
        prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI"]
    )
    {
        color3f[] primvars:displayColor = [(0.2784314, 0.4117647, 1)]
        double size = 25
        double3 xformOp:translate = (0, 0, 50)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
    def Cube "Box2" (
        prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI",
"PhysicsFilteredPairsAPI"]
    )
    {
        prepend rel physics:filteredPairs = </World/Box1>
        color3f[] primvars:displayColor = [(0.2784314, 0.4117647, 1)]
        double size = 25
        double3 xformOp:translate = (0, 0, 100)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
    def Cube "Ground" (
        prepend apiSchemas = ["PhysicsCollisionAPI"]
    )
    {
        color3f[] primvars:displayColor = [(0.5, 0.5, 0.5)]
        float3 xformOp:scale = (750, 750, 10)
        uniform token[] xformOpOrder = ["xformOp:scale"]
    }
    def SphereLight "SphereLight"
    {
        float intensity = 30000
        float radius = 150
        double3 xformOp:translate = (650, 0, 1150)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
}
```

Joint

```
#usda 1.0

#Shows a joint which is driven to rotate

#around the vertical axis with a constant

#speed.

(

    defaultPrim = "World"

    endTimeCode = 100

    metersPerUnit = 0.01

    startTimeCode = 0

    timeCodesPerSecond = 24

    upAxis = "Z"


    kilogramsPerUnit = 1.0

)

def Xform "World"

{

    def PhysicsScene "physicsScene"

    {

        float3 gravity = (0, 0, -1000)

    }

    def Cube "StaticBox" (

        prepend apiSchemas = ["PhysicsCollisionAPI"]

    )

    {

        color3f[] primvars:displayColor = [(0.64705884, 0.08235294, 0.08235294)]

        double size = 100

        quatf xformOp:orient = (1, 0, 0, 0)

        float3 xformOp:scale = (0.1, 1, 0.1)

        double3 xformOp:translate = (0, 0, 1000)

        uniform token[] xformOpOrder = ["xformOp:translate", "xformOp:orient",
"xformOp:scale"]

    }

    def Cube "DynamicBox" (

        prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI"]

    )

    {
```

```
        color3f[] primvars:displayColor = [(0.2784314, 0.64705884, 1)]

        double size = 100

        quatf xformOp:orient = (1, 0, 0, 0)

        float3 xformOp:scale = (0.1, 1, 0.1)

        double3 xformOp:translate = (0, 120, 1000)

        uniform token[] xformOpOrder = ["xformOp:translate", "xformOp:orient",
"xformOp:scale"]

    }

    #joint with 5 limits and one drive

    def PhysicsJoint "D6Joint" (

        prepend apiSchemas = ["PhysicsLimitAPI:transX", "PhysicsLimitAPI:transY",
"PhysicsLimitAPI:transZ", "PhysicsLimitAPI:rotX", "PhysicsLimitAPI:rotY",
"PhysicsDriveAPI:rotZ"]

    )

    {

        rel physics:body0 = </World/StaticBox>

        rel physics:body1 = </World/DynamicBox>

        float limit:rotX:physics:high = -1

        float limit:rotX:physics:low = 1

        float limit:rotY:physics:high = -1

        float limit:rotY:physics:low = 1

        float limit:transX:physics:high = -1

        float limit:transX:physics:low = 1

        float limit:transY:physics:high = -1

        float limit:transY:physics:low = 1

        float limit:transZ:physics:high = -1

        float limit:transZ:physics:low = 1

        float drive:rotZ:physics:targetVelocity = 10.0

        float drive:rotZ:physics:damping = 9999.0

        point3f physics:localPos0 = (0, 60, 0)

        point3f physics:localPos1 = (0, -60, 0)

        quatf physics:localRot0 = (1, 0, 0, 0)

        quatf physics:localRot1 = (1, 0, 0, 0)

    }

    def SphereLight "SphereLight"

    {

        float intensity = 30000

        float radius = 150
```

```
        double3 xformOp:translate = (650, 0, 1150)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
}
```

Distance Joint

```
#usda 1.0
#Shows a dynamic box connected
#to a fixed box with a distance joint.
(
    defaultPrim = "World"
    endTimeCode = 100
    metersPerUnit = 0.01
    startTimeCode = 0
    timeCodesPerSecond = 24
    upAxis = "Z"
    kilogramsPerUnit = 1.0
)
def Xform "World"
{
    def PhysicsScene "physicsScene"
    {
    }
    def Cube "StaticBox" (
        prepend apiSchemas = ["PhysicsCollisionAPI"]
    )
    {
        color3f[] primvars:displayColor = [(0.64705884, 0.08235294, 0.08235294)]
        double size = 100
        quatf xformOp:orient = (1, 0, 0, 0)
        float3 xformOp:scale = (0.1, 1, 0.1)
        double3 xformOp:translate = (0, 0, 1000)
        uniform token[] xformOpOrder = ["xformOp:translate", "xformOp:orient",
"xformOp:scale"]
    }
    def Cube "DynamicBox" (
        prepend apiSchemas = ["PhysicsCollisionAPI", "PhysicsRigidBodyAPI"]
    )
```

```
    {
        color3f[] primvars:displayColor = [(0.2784314, 0.64705884, 1)]
        double size = 100
        float3 velocity = (0, 0, 0)
        quatf xformOp:orient = (1, 0, 0, 0)
        float3 xformOp:scale = (0.1, 1, 0.1)
        double3 xformOp:translate = (0, 120, 1000)
        uniform token[] xformOpOrder = ["xformOp:translate", "xformOp:orient",
"xformOp:scale"]
    }
    def DistancePhysicsJoint "DistanceJoint"
    {
        rel physics:body0 = </World/StaticBox>
        rel physics:body1 = </World/DynamicBox>
        float3 physics:localPos0 = (0, 60, 0)
        float3 physics:localPos1 = (0, -60, 0)
        quatf physics:localRot0 = (1, 0, 0, 0)
        quatf physics:localRot1 = (1, 0, 0, 0)
        float physics:maxDistance = 50
        float physics:minDistance = 10
    }
    def SphereLight "SphereLight"
    {
        float intensity = 30000
        float radius = 150
        double3 xformOp:translate = (650, 0, 1150)
        uniform token[] xformOpOrder = ["xformOp:translate"]
    }
}
```