This document is best viewed with this option in the menubar: *View → Text Width → Wide*

Unity Physics 101

In this document, we'll first explain fundamental concepts of game physics, and then we'll cover essentials of the Unity Physics API. Lastly, we'll walk through a handful of very simple samples which demonstrate these essentials.

What does a physics engine do?

Before getting into the specifics of the Unity Physics package, we'll start with a general discussion of what physics engines typically do and how they work.

Every update, a physics engine does four main steps, in this order:

1. **Build a spatial data structure** of some kind that allows for efficient "intersection tests". An intersection test checks whether the geometries of two physics objects touch or overlap.
2. **Detect collisions** and generate *constraints* from the contact points between all touching or overlapping physics objects.

3. **Run the physics “solver”** to update the physics objects' velocities based on their constraints and other factors, such as their masses, gravity, and friction.
4. **Move the physics objects** based on their new velocities.

In your game code, the spatial data structure built by the physics engine may also be used for various purposes of game logic. For example, when a gun is fired, your game code can query the spatial structure to determine what the bullet might hit along its path.

> **Note**: Because the delta time between frames often varies and because the physics simulation shouldn’t necessarily update at the same rate as the rendering or game simulation, a physics simulation is usually updated at its own fixed update rate. This is achieved by updating the physics simulation each frame 0 or more times to approximate an average rate over time. For example, one frame may perform 0 physics updates, but the next frame may perform 3, and the one after that just 1, *etc*, with the aim of averaging out over time to match the intended physics update rate.

Rigid bodies

In most game physics engines, the physics objects are usually *rigid bodies*, meaning they are solid objects with uniformly distributed mass which do not bend, squish, or change shape in any way. This excludes *soft bodies* (deformable bodies like rubber), fluids, gasses, and particles like sand.

> **Note**: You might think rope, wire, cords, and cloth are deformable soft bodies, but they actually can be approximated as rigid body segments held together by constraints (see “joints” below). Often, however, these objects are handled by a separate, special solver for better efficiency and accuracy.

> **Note**: Some rigid body simulations also support destructible objects, allowing the objects to fracture and break apart.

Typically, the data for each rigid body object includes these elements:

- position
- orientation
- collider (the geometry used in collision detection)
- linear velocity (direction and speed of travel)
- angular velocity (direction and speed of rotation)
- mass
- mass distribution (inertia tensor: the rotational semi-equivalent of mass)
- center of mass (a point, usually inside the object)

...plus other optional elements:

- linear and angular damping (decay of velocities)
- gravity (per-object override of the global gravity)
- friction coefficient
- restitution (bounciness)
- collision filters

Colliders

The following collider shapes are supported in Unity Physics:

- **Box**: A rectangular prism defined by a length, height, and width.
- **Sphere**: A sphere defined by a radius.
- **Capsule**: A cylinder with round ends, defined by an inner line segment and a radius.
- **Cylinder**: A cylinder defined by a height and radius.
- **Triangle / Quad**: 3 or 4 coplanar vertices forming either a triangle or a quad.
- **Convex hulls**: An arbitrary convex hull defined by a set of 3D points.
- **Mesh**: A mesh composed of triangles and quads.
- **Compound**: A composite of multiple collider shapes.
- **Terrain**: A uniform grid of height samples.

Collision detection for mesh colliders is much more computationally expensive than for primitive colliders, such as spheres and boxes. Convex hull colliders are also more expensive than primitives but are still considerably cheaper than mesh colliders. The cost of mesh colliders and convex hull colliders scales with their geometric complexity.

For performance and gameplay reasons, an object's collider (*a.k.a.* collision geometry) very often differs from its visible geometry. For example, using the full animated mesh of a character for collision detection can have a large performance cost, and such shapes can easily get caught on the environment as the character tries to move around. Therefore, animated characters are often instead given simple shape colliders, such as capsules, which make the collisions much cheaper and less likely to get caught on the environment.

> **Note**: In some games, you might want characters to have accurate collisions for specific purposes, such as hit detection with bullets. In these scenarios, you can use a simple collider for character movements but also use a second, invisible, more detailed collider which moves with the character but which only collides with bullets. Such setups are also common use cases for collision filters (discussed later).

Compound colliders

A compound collider is a single logical collider formed from multiple colliders. The combined, resulting shape need not be convex, and the separate shapes need not touch or intersect.

Compound colliders are useful when:

- …you want a complex collider shape but want cheaper collisions than the equivalent mesh collider.
- …you want to distinguish between collisions for different parts of the object. For example, for a character, you want to distinguish between bullet collisions with the head, torso, or other body parts.

Dynamic and kinematic bodies

A *dynamic* body is a “normal” rigid body physics object, meaning a body affected by gravity and forces generated from its collisions.

A *kinematic* body is a rigid body with conceptually infinite mass. As such, its transform and velocity are not automatically affected by collisions or other interactions with other rigid bodies. Instead, the user’s own code is responsible for setting a kinematic body's transform and velocity to control how it moves.

> **Note**: According to Wikipedia, “Kinematics is a subfield of physics and mathematics, developed in classical mechanics, that describes the motion of points, bodies (objects), and systems of bodies (groups of objects) without considering the forces that cause them to move.” So in a physics engine, the term “kinematic” makes sense for an object that does not (automatically) have forces applied to it.

Character controllers are a key use case for kinematic bodies. Though dynamic body character controllers are used in some games, they can make it difficult for the programmer to exert control: when the programmer directly sets a dynamic body’s position or velocity, they're effectively overriding the force of any collisions applied by the physics engine, leading to scenarios where the object could phase through other colliders and other undesirable behaviors. On the other hand, a kinematic controller does not automatically factor in collisions with other colliders, so the programmer must manually factor these collisions into the controller logic when updating the body's velocity.

More generally, kinematic bodies can be useful when you want a moving object that affects other objects when it collides with them but which itself does not automatically react to those collisions or other physical properties (*e.g.* gravity, friction, or damping). For example, an elevator should move other objects resting on its floor, so you need it to be a physics object; however, you generally want to control an elevator's movements strictly in your own code rather than allow the elevator to get stuck on obstructions or be affected by gravity, so you should make the elevator a kinematic body.

Static colliders

A static collider is an object that has a collider but no velocity, mass, or other properties of a dynamic body. Generally, the objects that make up a game environment, such as static props, floors, walls, or terrains, should be static colliders.

In collisions between a static collider and dynamic or kinematic bodies, a static collider is treated as if it has infinite mass. In other words, a static collider is treated as an immovable object.

The physics engine does not check for collisions between static colliders.

> **NOTE**: In Unity Physics, it's possible to move static colliders at runtime, but doing so triggers an expensive rebuild of the static collider bounding volume hierarchy used for collision detection. Generally you should avoid moving static colliders. For a collider that moves occasionally, consider using a kinematic with 0 velocity.

Collision queries

It's often useful in your code to ask the physics engine questions like, "What would this object hit if it moved to this position or moved in this direction?” or, “What would a projectile hit if it were shot from here in this direction?", and so forth. Such questions are called *collision queries*.

The Unity Physics API provides five kinds of collision queries:

- **Overlap query**: Find all the colliders whose bounding boxes overlap a given shape.
- **Ray cast query**: Find the intersections, if any, between any of the colliders and a given ray (a directed line segment).
- **Collider cast query** (*a.k.a.* "shape cast" or "sweep"): Find the intersections, if any, between the colliders and a given shape which is moved along a given line segment.
- **Collider distance query**: Find the shortest line segment connecting a given shape and each collider within a given maximum radius.
- **Point distance query**: Find the shortest line segment connecting a given point and each collider within a given maximum radius.

Collision and trigger events

For detected collisions between physics objects, the physics engine may generate *collision events* with information about the collisions. Processing these events in your code can be useful for many gameplay purposes, such as allowing your kinematic character controller to respond to collisions.

Because raising events for *every* collision could be very costly, physics engines typically only raise collision events for bodies designated by the user to raise events.

A *trigger* is a (usually invisible) collider whose collisions do not generate constraints, so other bodies can intersect the trigger freely.

For detected collisions between a physics object and a trigger, the physics engine generates *trigger events* with information about the collisions. These events then can then be read in your code to watch for occurrences such as the player character walking into a certain area.

Collision filtering

In many gameplay scenarios, you don't want every physics object to necessarily generate contact constraints and events when they collide with every other. For example, maybe you want projectiles in your game to pass through cosmetic debris, or maybe some moving objects that are tethered together should pass through each other.

To solve these problems, collision filters let you specify which individual objects or groups of objects should generate contact constraints and events when they collide with each other.

The broad phase and narrow phases of collision detection

Because it can be quite costly to determine which pairs of physics bodies collide with each other, collision detection is usually split into two phases:

1. In the first phase, called the "broad" phase, the physics engine finds the colliders whose Axis-Aligned Bounding Boxes (AABBs) intersect. An AABB is a box which is aligned with the world axes (in other words, a box which is not rotated) and which fully surrounds the object.
2. In the second phase, called the "narrow" phase, the pairs of objects whose AABBs intersected in the broad phase are checked to see if their actual colliders intersect.

Because AABB intersection tests are individually cheap (especially compared to hull and mesh intersection tests), splitting collision detection into broad and narrow phases typically reduces the number of pairwise intersection tests required for collision detection.

The broad phase is also often further accelerated by grouping the AABB's of the colliders into some kind of spatial structure, such as an octree or Bounding Volume Hierarchy (BVH), thus allowing the broad phase to only check for intersections between AABBs in the same part of the world.

Unity Physics creates a BVH for the dynamic bodies and kinematics, plus a separate additional BVH for static colliders.

> **NOTE**: Be careful about large colliders: colliders with very large AABB's will often end up intersecting many other objects in the broad phase, possibly leading to an excessive number of expensive pairwise intersection tests in the narrow phase. In such cases, consider breaking the large collider into smaller pieces.

Joints and motors

In addition to the contact constraints generated by the physics engine's collision detection, you may sometimes wish to manually create additional constraints that last indefinitely. For example, you may wish to tether two bodies together such that they drag each other when moved. These persistent constraints between two bodies are called *joints*.

Depending upon a joint's configuration, its constraints may restrict certain degrees of freedom (DoF), *e.g.* a joint might only allow relative rotation of the two bodies along a single axis.

Unity Physics offers several kinds of joints:

- **Ball and Socket**: Allows motion around an indefinite number of axes. Humans have such joints in the hips and shoulders.
- **Limited Hinge**: Allows limited articulation on one axis. Humans have such joints in the fingers and knees.
- **Fixed**: Constrains two rigid bodies together, removing their ability to move independently of each other.
- **Hinge**: Allows free rotation on one axis. Can be used for spinning wheels and carousels.
- **Prismatic**: Constrains two bodies to a sliding motion on one axis. Can be used to make various sliding doors.
- **Ragdoll**: Limits the motion on a few axes. Useful for creating characters.
- **Stiff Spring**: Constrains two bodies to be a certain distance apart from each other.

A *motor* is a variant of joint which has a "driven" constraint, meaning a constraint which induces force on the two bodies even when they otherwise are at rest.

Unity Physics offers a few kinds of motors:

- **Rotation Motor**: A motorized hinge joint that rotates about an axis toward a target angle.
- **Angular Velocity Motor**: A motorized hinge joint that rotates about an axis at a constant target velocity.
- **Position Motor**: A motorized prismatic joint that drives toward a relative target position.
- **Linear Velocity Motor**: A motorized prismatic joint that drives to a relative target position at a constant target velocity.

Unity Physics API

Now let's get into the specifics of the Unity Physics package.

Static collider entities

A static collider entity represents collision geometry that is generally expected not to move, such as the ground and walls that make up a typical game environment. Moving a static collider at runtime triggers an expensive rebuild of the static collider hierarchy, so it's generally recommended that you move static colliders only rarely or not at all.

Static collider entities are defined by two components:

- **PhysicsCollider**: The core info about the collider, which includes:
    - The collider geometry
    - Collision filter properties
    - Mass distribution properties
    - Friction properties
    - Restitution (bounciness) properties

    Note: The collision geometry is stored in a BlobAssetReference<Collider>. Because PhysicsCollider components of different entities may share the same BlobAssetReference<Collider>, changes to an individual collider may affect the colliders of multiple entities. To avoid this, you may need to give entities their own separate copies of a collider.

- **PhysicsWorldIndex**: A shared component that denotes which “physics world” this entity belongs to. Physics entities only collide and interact with other physics entities of the same physics world.

  > **Note:** Bee clear that the concept of “physics world” is separate from the general Entities concept of entity Worlds. Whereas an entity World contains a set of entities and systems that is logically isolated from other entity Worlds, a physics world contains a subset of the physics entities *within* a single entity World. Usually, only the default physics world (0) is needed, but additional physics worlds can be useful in advanced cases.

A compound collider (a collider made out of multiple separate pieces of geometry) may also have this component:

- **PhysicsColliderKeyEntityPair**: A buffer that references the colliders that make up a compound collider.

Rigid body entities

A dynamic rigid body entity represents a physics body with velocity, mass, and other physical properties.

A kinematic rigid body is a dynamic rigid body with infinite mass and inertia which therefore does not automatically respond to collision forces. Kinematic rigid bodies are commonly used for character and vehicle controllers, as well as other things that might be programmatically moved, like say an elevator or moving platform.

Rigid body entities have the components of a collider plus also:

- **PhysicsVelocity**: The linear and angular velocity of the body.
- **PhysicsMass**: The mass, center of mass, inertia, and angular expansion factor.

  > **Note:** The linear velocity of a PhysicsMass is expressed in world space, but the angular velocity is expressed relative to the PhysicsMass transform rather than world space. The “angular expansion factor” determines how much to expand a rigid body's AABB to enclose its swept volume.

Optional components that define additional physical properties:

- **PhysicsDamping**: Damping applied to the velocities in each simulation step.
- **PhysicsGravityFactor**: A gravity factor that modifies the global gravity for this body.

- **PhysicsCustomTags**: User-customizable data which is copied to any collision or trigger events involving this body. Useful for classifying bodies in your event-processing code.
- **PhysicsTemporalCoherenceInfo**:
- **PhysicsTemporalCoherenceTag**:
- **PhysicsMassOverride**: Contains a flag that can toggle the entity between kinematic and non-kinematic. Not required for kinematic rigid body entities unless they need to toggle between kinematic and non-kinematic.

Optional components related to graphics:

- **PhysicsRenderEntity**: Sometimes you may wish to express a physical object and its graphical rendering as two separate entities. For a physics entity that isn't itself rendered, this component stores a reference to the entity that is meant to be its graphical representation.
- **PhysicsGraphicalSmoothing**: Specifies that the motion of the graphical representation entity should be smoothed (but only when the rendering framerate is greater than the physics fixed step rate).
- **PhysicsGraphicalInterpolationBuffer**: Stores the state of a rigid body from the previous physics tick in order to interpolate the motion of the body's graphical representation. When used in conjunction with PhysicsGraphicalSmoothing, this component indicates that smoothing should interpolate between the two most recent physics simulation ticks, which results in a more accurate representation of the physics simulation (but with an added tick of latency).

Joints and motors

A *joint* represents a set of constraints that connect two rigid body entities. A *motor* is a kind of joint with a "driving" constraint that applies forces on the two rigid bodies, such as a rotational spin force.

Each joint or motor is represented as an entity with these two components:

- **PhysicsJoint**: The properties of the joint:
    - Two BodyFrames (transforms which define the joint relative to the two bodies)
    - A JointType enum (denotes the type of joint: Fixed, Hinge, RotationalMotor, *etc.*)
    - A set of Constraints
- **PhysicsConstrainedBodyPair**
    - A pair of Entity ids
    - An int denoting whether the two bodies should generate collision events

Joint and motor entities that are part of a set may also have the following component:

- **PhysicsJointCompanion**: A buffer referencing the other joint entities that form a complex joint configuration

  Note: Joints and motors do not necessarily have to be their own separate entities (though they often are). For example, a joint can be created by adding the PhysicsJoint and PhysicsConstrainedBodyPair components to one of the two rigid body entities connected by the joint. All that really matters is that a PhysicsJoint and its related PhysicsConstrainedBodyPair (plus optional PhysicsJointCompanion) are placed on the same entity.

Additional singleton components

Two singleton components provide access to some core Unity Physics functionality:

- **SimulationSingleton**: Provides access to the Simulation, which is primarily used when you need to process collision, trigger, and impulse events.
- **PhysicsWorldSingleton**: Provides access to the PhysicsWorld.

Unity Physics will create entities with these two components automatically.

Two additional singleton components specify some global physics settings and enable some visual debugging tools:

- **PhysicsStep:** Contains properties that determine how the physics world is stepped. If no instance of this singleton exists, default values are used.
- **PhysicsDebugDisplayData:** Contains properties that determine whether to draw physics debug visualizations, such as wireframes of the colliders. If no instance of this singleton exists, default values are used.

Unity Physics does not create entities with these two components automatically.

> **Note:** The term “singleton” here means that these components should each be added to no more than one entity in the world.

Creating physics entities in the editor

> **Note:** This section assumes you have basic familiarity with entity baking.

To create physics entities in a subscene, you have three options:

1. Use the "**Built-in" physics components** (the standard GameObject physics components). Because these Built-in components were designed for PhysX, they don't neatly map to Unity Physics in all regards, but they are sufficient for most common cases.
2. Use the **physics authoring components published in the Unity Physics package samples**. These authoring components were designed for Unity Physics and so map more accurately to its functionality.
3. Make **your own custom authoring components**. If you study the authoring components from the package samples, you may learn enough to create your own that better fit your needs. Be warned, though, that some facets of the underlying Unity Physics component data (such as those related to joints and motors) require deep understanding of the physics engine to set up properly.

(For simplicity, we will focus on the first option in the samples below.)

When a GameObject with a collider component in a subscene is baked, the created entity is given the Unity.Physics components PhysicsCollider and PhysicsWorldIndex.

When a GameObject with a RigidBody component in a subscene is baked, the created entity is given the Unity.Physics components PhysicsVelocity, PhysicsMass, and PhysicsDamping.

Creating physics entities at runtime

To create a static collider entity at runtime:

1. Create an entity.
2. Add the transform components, LocalTransform and LocalToWorld, to the entity.
3. Add the rendering components to the entity by calling RenderMeshUtility.AddComponents.
4. Set the RenderBounds component with an AABB that encompasses the object.
5. Add a PhysicsWorldIndex component to the entity. (The default value of 0 is fine if you aren't using more than one physics world.)
6. Add a PhysicsCollider component to the entity. To create a BlobAssetReference<Collider> for the PhysicsCollider, you can use one of the convenience methods that return a collider:
    - BoxCollider.Create
    - CapsuleCollider.Create
    - CompoundCollider.Create
    - ConvexCollider.Create

    - CylinderCollider.Create
    - MeshCollider.Create
    - PolygonCollider.CreateTriangle
    - PolygonCollider.CreateQuad
    - SphereCollider.Create
    - TerrainCollider.Create

To create a dynamic body entity at runtime:

1. Follow the same steps as for a static collider.
2. Add a PhysicsVelocity component to the entity.
3. Add a PhysicsMass component to the entity. To create the PhysicsMass value, you can call PhysicsMass.CreateDynamic or PhysicsMass.CreateKinematic

See a full code example here.

To create a joint or motor entity at runtime:

1. Create an entity.
2. Add a PhysicsWorldIndex component to the entity. The default value of 0 is usually fine (unless you are using multiple physics worlds).
3. Add the PhysicsConstrainedBodyPair component to the entity. Set the component to reference the two body entities.
4. Add the PhysicsJoint component. There are a number of static methods for creating a PhysicsJoint with the appropriate settings:
    - PhysicsJoint.CreateAngularVelocityMotor
    - PhysicsJoint.CreateBallAndSocket
    - PhysicsJoint.CreateFixed
    - PhysicsJoint.CreateHinge
    - PhysicsJoint.CreateLimitedDistance
    - PhysicsJoint.CreateLimitedDOF
    - PhysicsJoint.CreateLimitedHinge
    - PhysicsJoint.CreateLinearVelocityMotor
    - PhysicsJoint.CreatePositionMotor
    - PhysicsJoint.CreatePrismatic
    - PhysicsJoint.CreateRagdoll
    - PhysicsJoint.CreateRotationalMotor

    Note: Manually setting the fields of the PhysicsJoint correctly requires deep understanding of the solver and so is not recommended for most users.

    Also note that, although the Built-in (GameObject PhysX) joint MonoBehaviours can be used to bake joint entities in subscenes, a few properties of these MonoBehaviours are ignored in baking because they do not map directly to attributes of a Unity Physics joint.

The Physics systems

This is the hierarchy of physics-related system groups and systems (as of Unity Physics version 1.3.8):

- **Simulation System Group**
    - **Fixed Step Simulation Group**
        - Inject Temporal Coherence Data System
        - **Physics System Group**
            - Sync Custom Physics Proxy System
            - Collider Blob Cleanup System
            - **Before Physics System Group**
                - Ensure Unique Collider System
            - **Physics Initialize Group**
                - Clean Physics Debug Data System_Default
                - **Physics Initialize Group Internal**
                    - Clean Physics World Dependency Resolver
                    - **Physics Build World Group**
                        - Inject Temporal Coherence Data Last Resort System
                        - Build Physics World
                        - Invalidated Temporal Coherence Cleanup System
                        - Integrity Check System
                    - Physics Analytics System
            - **Physics Simulation Group**
                - Physics Simulation Picker System
                - **Physics Create Body Pairs Group**
                    - Broadphase System
                - **Physics Create Contacts Group**
                    - Narrowphase System
                - **Physics Create Jacobians Group**
                    - Create Jacobians System
                - **Physics Solve And Integrate Group**
                    - Solve and Integrate System
            - Buffer Interpolated Rigid Bodies Motion
            - Record Most Recent Fixed Time
            - Export Physics World

                - **After Physics Simulation Group**
                    - Display Collision Events System
                    - Display Trigger Events System
                - Modify Joint Limits System
                - Copy Physics Velocity to Smoothing
    - **Transform System Group**
        - Smooth Rigid Bodies Graphical Motion
    - **Late Simulation System Group**
        - **Physics Debug Display Group**
            - Display Body Colliders System
            - Display Broadphase Aabbs System
            - Display Collider Aabbs System_Default
            - Display Body Collider Edges_Default
            - Query Tester System
            - Display Trigger Events System
            - Display Mass Properties System
            - Display Joints System_Default
            - Display Collision Events System
            - Physics Debug Display System_Default

> **Note**: When using Netcode for Entities, the server puts the PhysicsSystemGroup inside the PredictedFixedStepSimulationGroup.

The core physics update happens within each iteration of the PhysicsSystemGroup:

1. The PhysicsInitializeGroup sets up the “physics world”. This entails copying transforms, velocities, and other physics properties from the physics body entities to Unity Physics’s internal representation of the physics simulation.
2. The BroadphaseSystem identifies the pairs of bodies that have overlapping AABB’s.
3. The NarrowPhaseSystem determines which of these pairs actually collide and generates contacts accordingly.
4. The CreateJacobiansSystem creates jacobians from the contacts and joint constraints.
5. The SolveAndIntegrateSystem solves the jacobians to update the transforms and velocities within the physics world.
6. The ExportPhysicsWorld system copies the transforms and velocities from the physics world back to the physics body entities.

Keep in mind that PhysicsSystemGroup is inside FixedStepSimulationGroup, which iterates 0 or more times each frame based on the fixed step rate, so 0 or more physics updates may be performed each frame.

> **Note**: ExportPhysicsWorld expects the rigid bodies to remain in the same chunk location as found by the PhysicsInitializeGroup, so it is not safe to make structural changes in between that would move these entities or destroy them (*e.g.* do not add components or

remove components from these entities). In the editor and development builds, ExportPhysicsWorld performs “integrity checks” to catch such mistakes, but the checks are skipped in release builds. Also understand that, because ExportPhysicsWorld overwrites the values of the physics components, any writes you make directly to these components in between PhysicsInitializeGroup and ExportPhysicsWorld will be clobbered and effectively ignored.

To order a system relative to the physics updates, you’ll typically use one of these attributes:

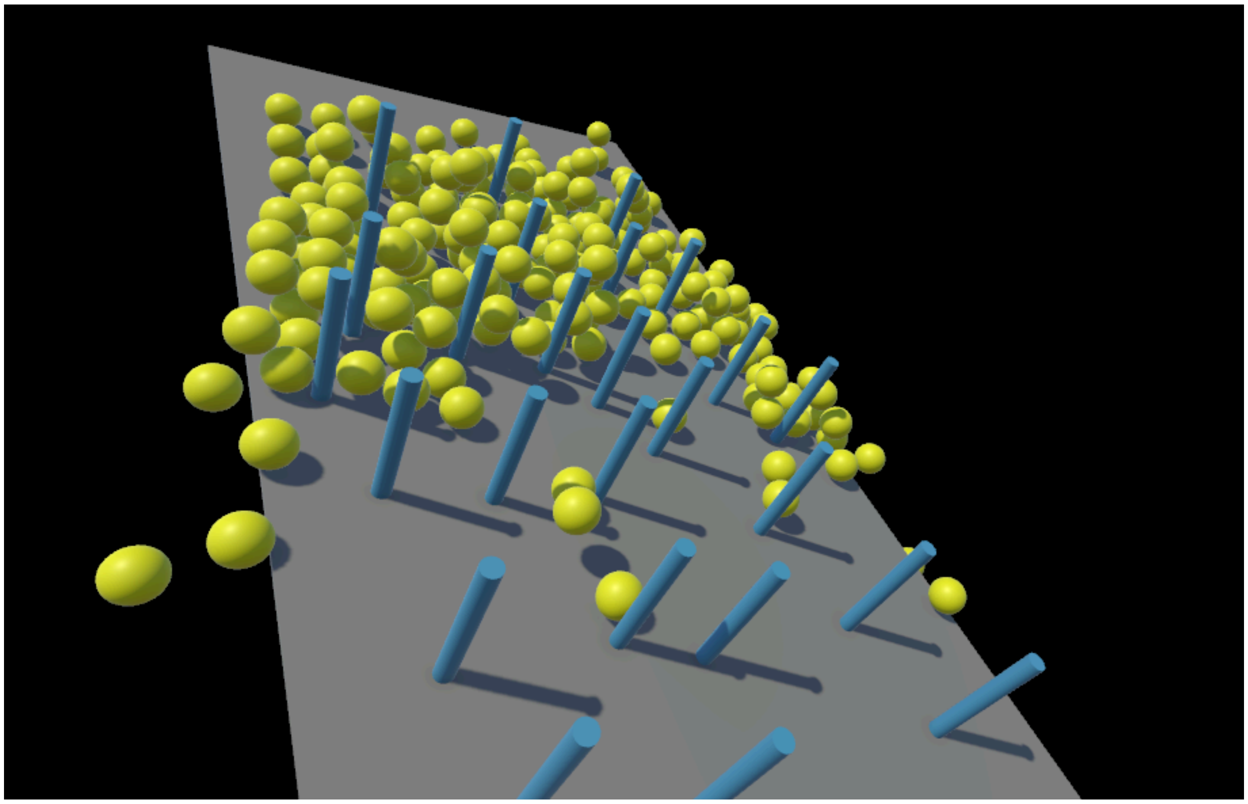
- [UpdateBefore(typeof(FixedStepSimulationGroup))]: Update once in the frame before all physics updates (even if none)
- [UpdateAfter(typeof(FixedStepSimulationGroup))]: Update once in the frame after all physics updates (even if none)
- [UpdateInGroup(typeof(BeforePhysicsSystemGroup))]: Update immediately before each physics update of the frame (if any)
- [UpdateInGroup(typeof(AfterPhysicsSystemGroup))]: Update immediately after each physics update of the frame (if any)

Physics Samples

The Unity project containing these samples is found here.

Sample: Pachinko Machine

In the first sample, some yellow balls (rigid bodies) fall through a pachinko board (composed of static colliders).



To make the colliders, each GameObject of the subscene has one of the standard GameObject collider components:

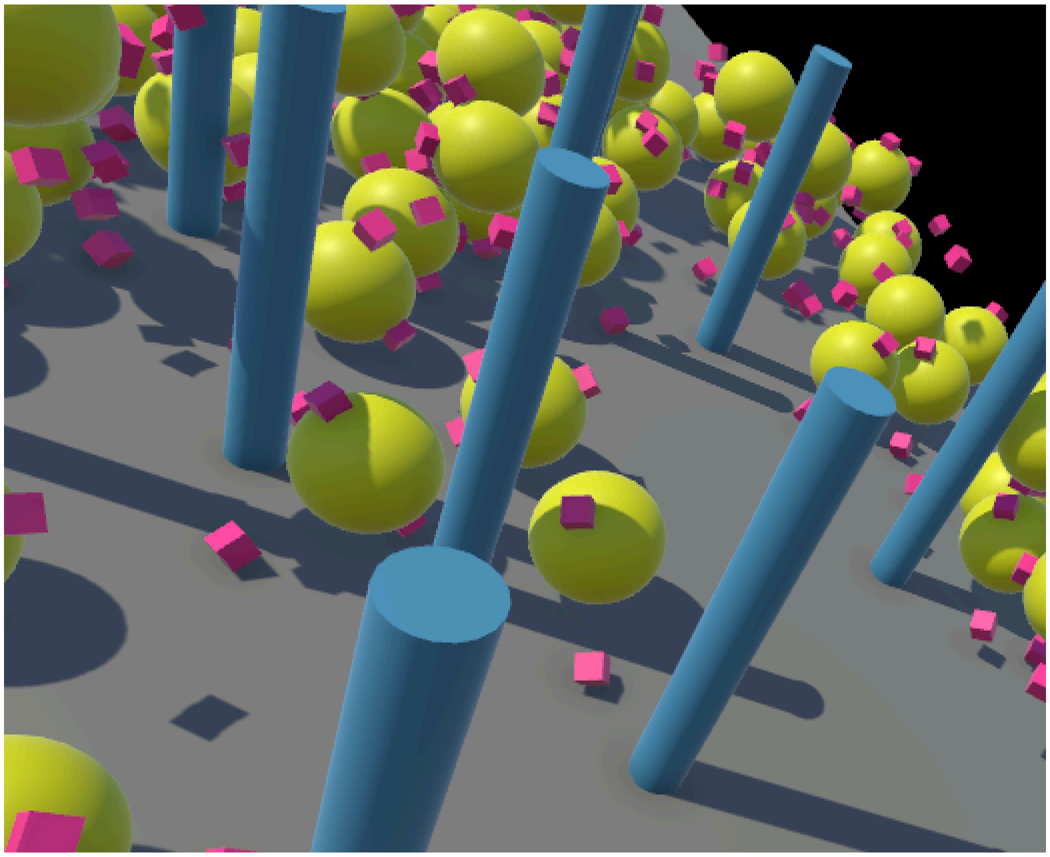
- The balls have a UnityEngine.SphereCollider component.
- The pegs have a UnityEngine.CapsuleCollider component.
- The board has a UnityEngine.BoxCollider component.

To make the balls into dynamic bodies, each has a UnityEngine.Rigidbody component.

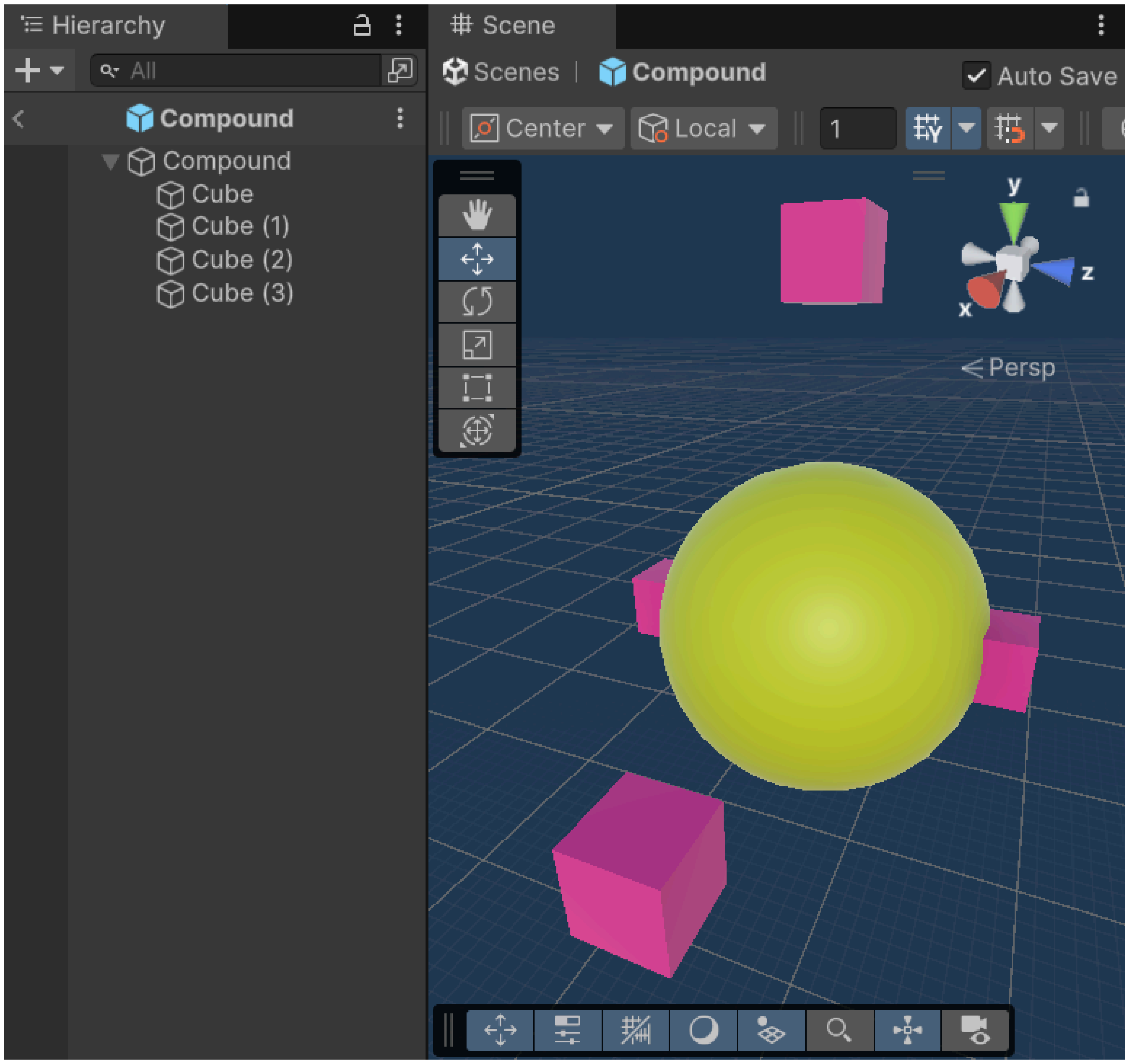
When the subscene is baked, the entities produced will each have PhysicsCollider and PhysicsWorldIndex components, and the balls will also have PhysicsVelocity, PhysicsMass, and PhysicsDamping components.

> **Note**: Again, we could instead use the custom authoring components from the Unity Physics samples or make our own custom authoring components, but for most simple cases, like these, the Built-in GameObject PhysX components work fine.

The sample has two additional variants. In one variant, the falling rigid bodies have compound colliders (a few pink boxes surround the sphere):

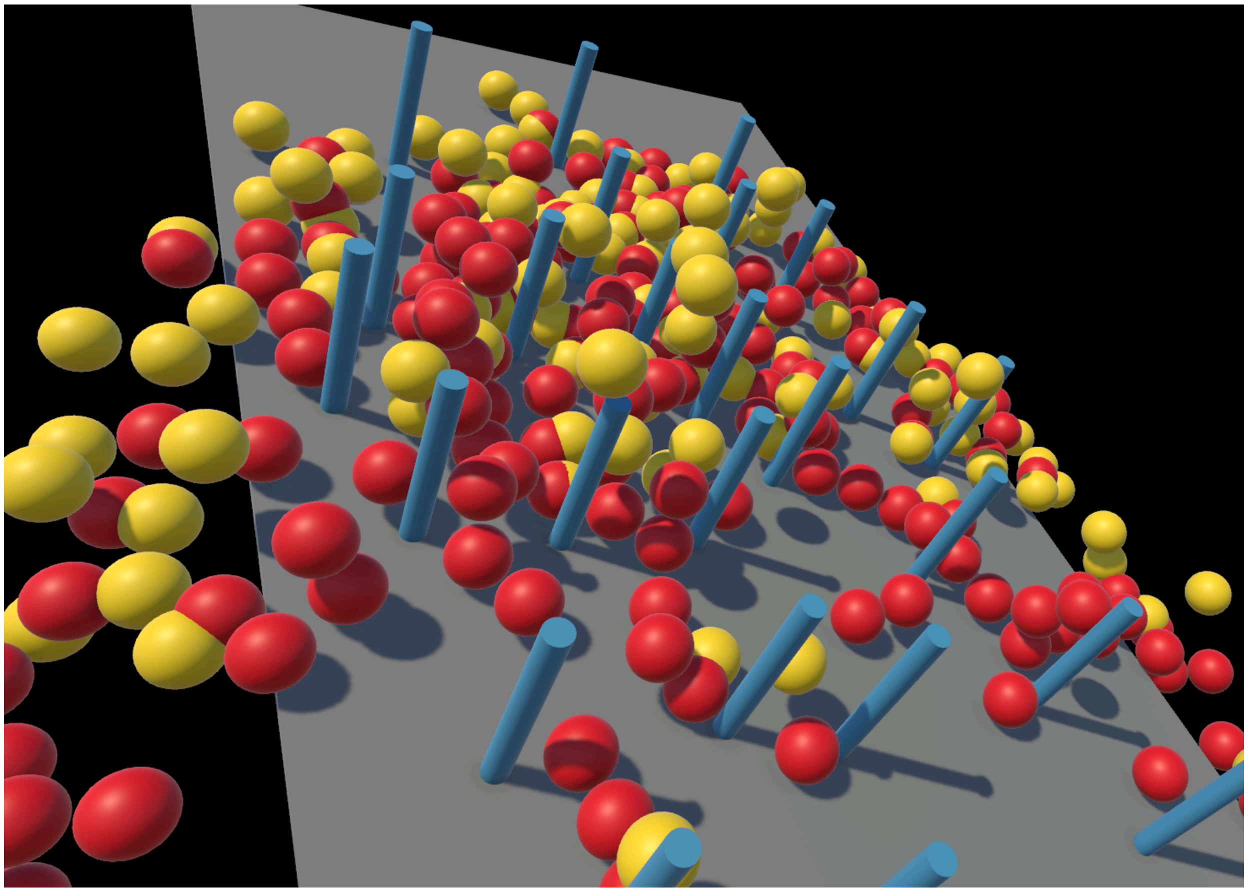


To create these compounds, the root ball GameObject has a child box GameObject, each with its own UnityEngine.BoxCollider. The root itself retains the UnityEngine.SphereCollider and the UnityEngine.Rigidbody.



When each ball-box compound is baked, the children's box colliders are "lifted" to form a compound with the sphere collider. In the end result, each ball-box compound is represented by 5 entities: a rendered sphere with a compound collider; and 4 additional rendered boxes which each have a transform parented to the sphere but no colliders of their own.

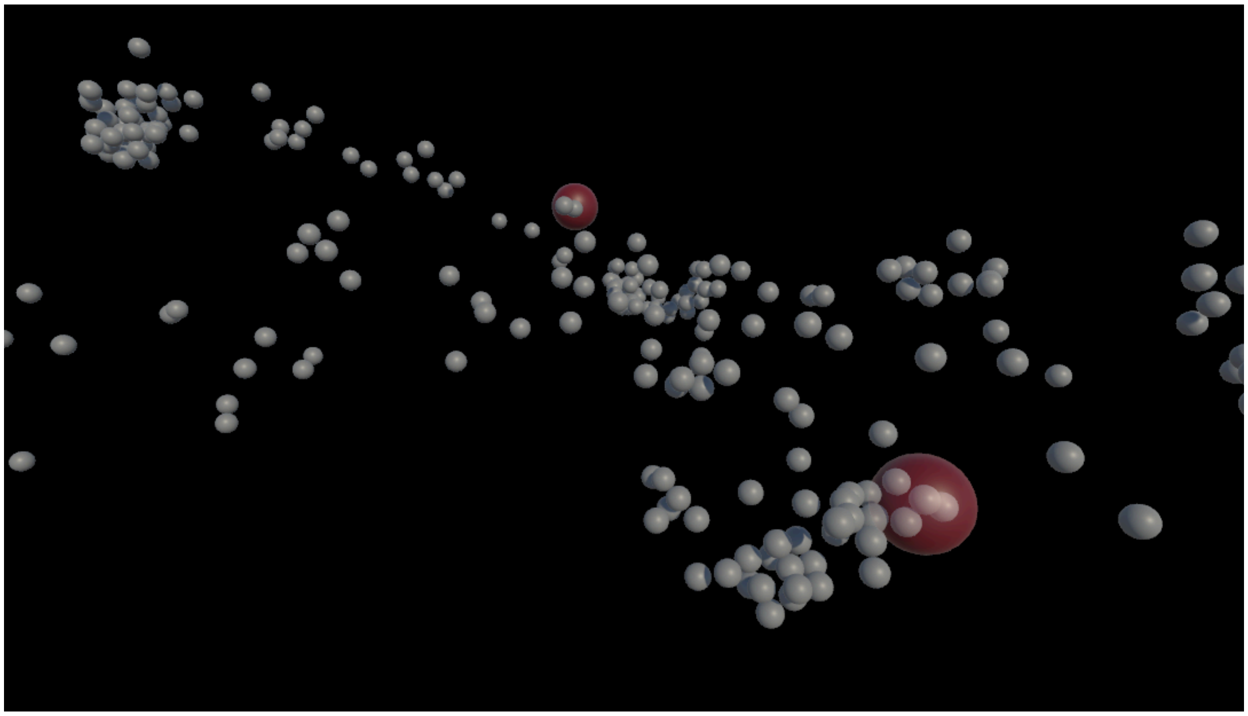
In the other variant of the pachinko machine, red balls and yellow balls spawn, but the red balls do not collide with the yellow balls, thanks to collision filtering:



To achieve this collision filtering, the red balls are added to a GameObject layer called “Red”, and the yellow balls are added to a GameObject layer called “Yellow”. In the UnityEngine.SphereCollider component, the LayerOverrides → ExcludeLayers property of the red balls is set to “Yellow”, and the property of the yellow balls is set to “Red”. When these GameObjects are baked, the PhysicsCollider components of the resulting entities will have the appropriate collision filter settings.

Sample: Gravity Well

In this sample, two red gravity wells orbit around the origin, and white balls gravitate towards the gravity wells. The balls will collide with each other, but the wells have no colliders.



In the main subscene, a Config singleton IComponentData is added by the ConfigAuthoring MonoBehaviour. This config contains configuration parameters:

- The number of balls to spawn.
- The prefab for the balls.
- The orbiting radius of the gravity wells.
- The orbiting speed of the gravity wells.
- The attraction force of the gravity wells.

The ball entities are spawned in a grid arrangement by the BallSpawnSystem. Each ball has a Ball IComponentData added in baking by a BallAuthoring MonoBehaviour. Because we don't want the balls to fall down, the UnityEngine.Rigidbody property "Use Gravity" is false.

Both gravity well entities are created in the subscene, and both have a GravityWell IComponentData added in baking by a GravityWellAuthoring MonoBehaviour. Each GravityWell component contains a float “OrbitalPos”, which represents the well's current position in radians as it travels in a circle around the origin.

Every frame, the GravityWellSystem moves the gravity wells around the origin and applies an impulse on the balls towards both gravity wells. GravityWellSystem updates in the AfterPhysicsSystemGroup, which is generally the appropriate point in the frame to modify properties of physics objects. The force is applied using ApplyExplosionForce, an extension method of PhysicsVelocity:

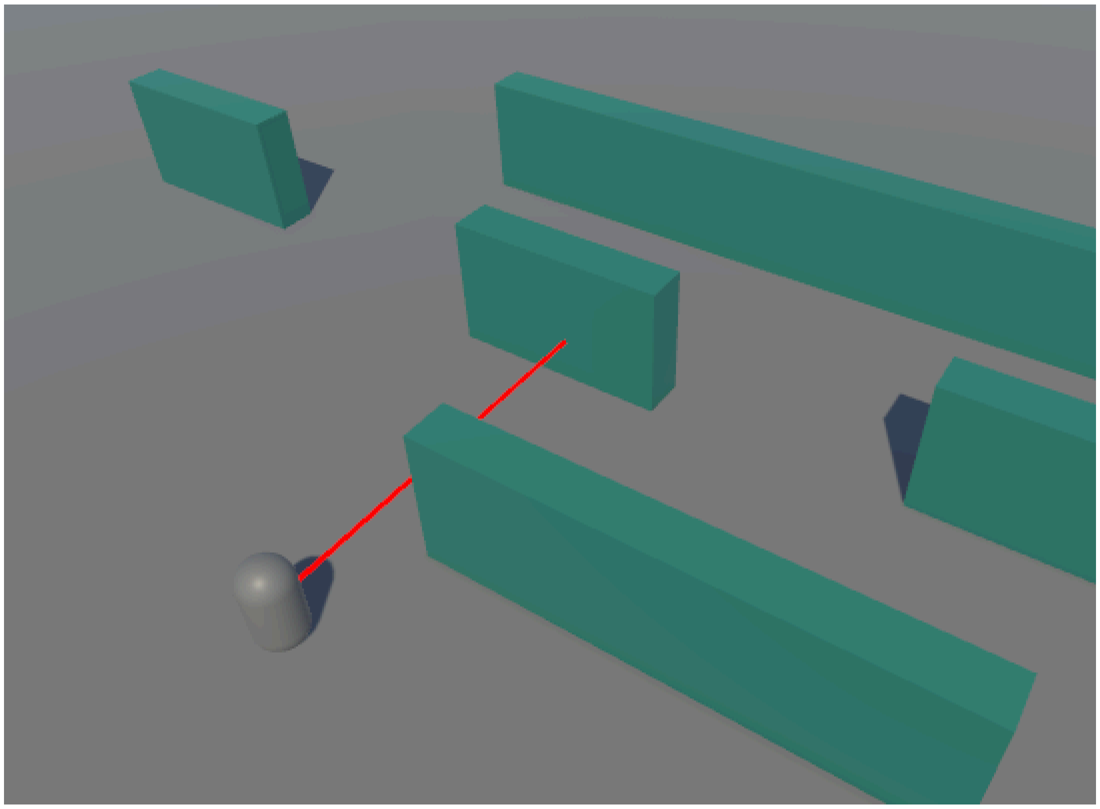
```
Java
// for each ball
foreach (var (velocity, collider, mass, ballTransform) in
        SystemAPI.Query<RefRW<PhysicsVelocity> RefRO<PhysicsCollider>,
RefRO<PhysicsMass> RefRO<LocalTransform>>())
{
        // for each gravity well, apply force towards the well
        for (int i = 0; i < wellTransforms.Length; i++)
        {
                var wellTransform = wellTransforms[i];

                velocity.ValueRW.ApplyExplosionForce(
                        mass.ValueRO, collider.ValueRO,
                        ballTransform.ValueRO.Position,
                        ballTransform.ValueRO.Rotation,
                        -config.WellStrength,    // negative strength makes this an
                implosion
                        wellTransform.Position,
                        0,    // explosion radius (0 makes reach infinite with no
                fall off over distance)
                        deltaTime,
                        math.up());
        }
}
```

Note: While we could set the balls' velocities and transforms directly, doing so would effectively nullify any other forces applied by the collisions and other factors. By applying a force to the velocity, we are adding additional force without nullifying other forces or nullifying the existing momentum.

Sample: Laser Sight

In this sample, a player character points a red laser sight that extends to the nearest wall.

The walls are simple box colliders. The player is a capsule with no collider that is moved by player input. For simplicity, the laser is pointed in a fixed direction from the player.



The code is all in the LaserSystem, which:

1. Moves the player character
2. Does a raycast to determine the laser length (the distance from the player to the nearest wall in the aiming direction)
3. Sets the laser endpoints

The physics-relevant part is the raycast:

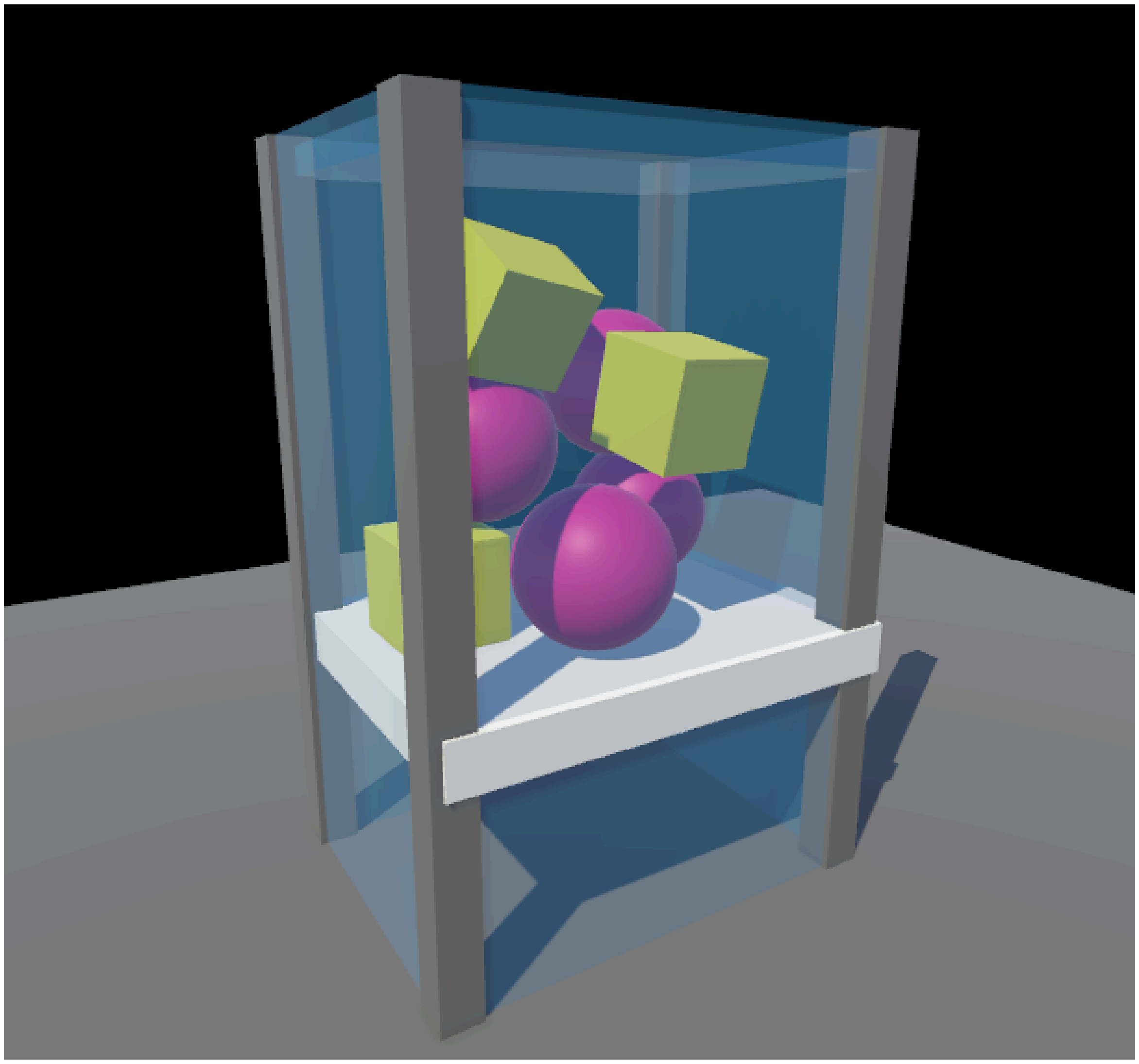
```java
// to perform raycasts or other collision queries, we need the collision world
var collisionWorld =
SystemAPI.GetSingleton<PhysicsWorldSingleton>().CollisionWorld;

foreach (var (playerTransform, player) in
        SystemAPI.Query<RefRO<LocalTransform>, RefRW<Player>>())
{
        var raycast = new RaycastInput
        {
                // specify starting and end points of the raycast (which
        together imply direction)
```

```
            Start = playerTransform.ValueRO.Position,
            End = playerTransform.ValueRO.Position + new float3(0, 0,
        config.MaxLaserLength),
            // don't forget to set a filter or else you'll get no hits!
            Filter = CollisionFilter.Default
        };

        if (collisionWorld.CastRay(raycast, out var closestHit))
        {
            // set laser length to the distance of the closest hit
            player.ValueRW.LaserLength =
        math.distance(playerTransform.ValueRO.Position, closestHit.Position);
        }
        else
        {
            // no hit detected, so just set the laser to max length
            player.ValueRW.LaserLength = config.MaxLaserLength;
        }
}
```

Sample: Elevator

In this sample, an elevator platform moves up and down inside a glass box. Several dynamic body shapes rest on top of the elevator platform.

When the elevator reaches the top or bottom of the box, it reverses direction. At these extremes, the shapes end up phasing through the elevator platform (which is simply how the physics solver resolves the contrary forces applied to the boxes).



Because we want to control the platform's movements in code while having other rigid bodies respond to collisions with the platform, the elevator platform rigid body is made kinematic (by checking the "kinematic" checkbox in the UnityEngine.Rigidbody).

The elevator platform velocity is modified by the ElevatorSystem:

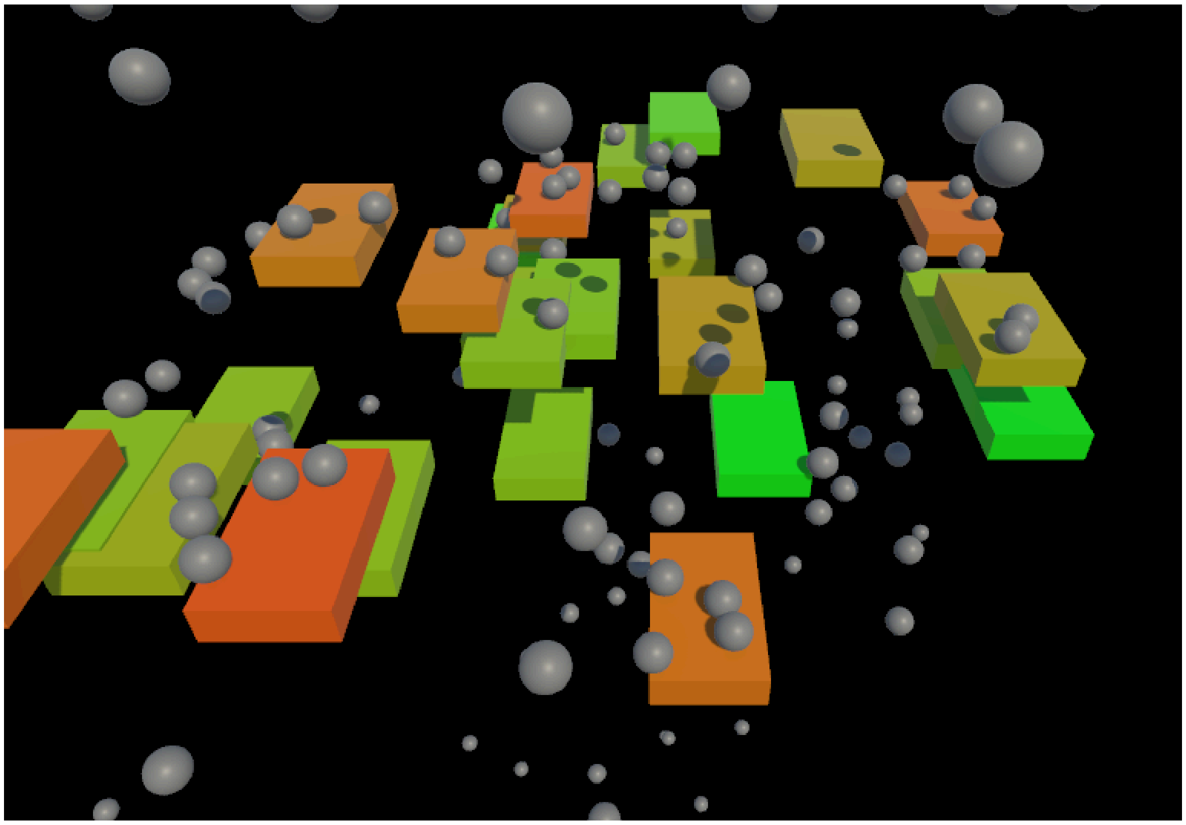
```java
foreach (var (elevator, trans, velocity) in
       SystemAPI.Query<RefRW<Elevator>, RefRO<LocalTransform>,
RefRW<PhysicsVelocity>>())
{
       // if not moving...
       if (velocity.ValueRW.Linear.y == 0)
```

```
        {
            // go up
            velocity.ValueRW.Linear.y = elevator.ValueRO.Speed;
        }
        // if going up...
        else if (velocity.ValueRW.Linear.y > 0)
        {
            // if hit top...
            if (trans.ValueRO.Position.y > elevator.ValueRO.MaxHeight)
            {
                // go down
                velocity.ValueRW.Linear.y = -elevator.ValueRO.Speed;
            }
        }
        // if going down...
        else
        {
            // if hit bottom...
            if (trans.ValueRO.Position.y < elevator.ValueRO.MinHeight)
            {
                // go up
                velocity.ValueRW.Linear.y = elevator.ValueRO.Speed;
            }
        }
    }
```

Note: Generally, you should modify the velocity of a kinematic rather than update the transform directly, unless you need to actually teleport the kinematic.

Sample: Breaking Bricks

In this sample, the impact of falling balls damages bricks. The brick color indicates the remaining ‘hit points’ of the brick, and a brick disappears when it runs out of hit points.



The BallSystem spawns the balls (simple dynamic bodies) at random locations above the bricks and then destroys the balls when they fall below a certain elevation.

The BrickSystem spawns the bricks at random locations, reduces the bricks' hit points when they are struck by a ball, and destroys the balls when their hit points reach 0. The amount of hit points deducted is proportional to the force of the impact. Here's the relevant code:

```Java
// ... inside BrickSystem

// get the collision events
var physicsWorld =
SystemAPI.GetSingleton<PhysicsWorldSingleton>().PhysicsWorld;
var sim = SystemAPI.GetSingleton<SimulationSingleton>().AsSimulation();

// to access the collisions events on main thread, we must sync any
outstanding physics sim jobs
sim.FinalJobHandle.Complete();

const float minImpactThreshold = 2f; // ignore impacts below this (to
effectively ignore resting contacts)
```

```
var strengthModifier = config.ImpactStrength;

var ecb = new EntityCommandBuffer(Allocator.Temp);

foreach (var collisionEvent in sim.CollisionEvents)
{
       // check if one of the two bodies is a brick and the other a ball
       Entity brickEntity;
       Entity ballEntity;

       // Note that, in a single physics update, a collision between a pair
       // of bodies creates one collision event, not two.
       // The API makes no guarantee which body is EntityA and which is
EntityB,
       // so you must test for both possibilities.

       if (SystemAPI.HasComponent<Brick>(collisionEvent.EntityA) &&
              SystemAPI.HasComponent<Ball>(collisionEvent.EntityB))
       {
              brickEntity = collisionEvent.EntityA;
              ballEntity = collisionEvent.EntityB;
       }
       else if (SystemAPI.HasComponent<Brick>(collisionEvent.EntityB) &&
              SystemAPI.HasComponent<Ball>(collisionEvent.EntityA))
       {
              brickEntity = collisionEvent.EntityB;
              ballEntity = collisionEvent.EntityA;
       }
       else
       {
              // skip because the collision is not between a brick and ball
              continue;
       }

       var details = collisionEvent.CalculateDetails(ref physicsWorld);

       // ignore resting contacts
       if (details.EstimatedImpulse < minImpactThreshold)
       {
              continue;
       }

       // reduce brick hitpoints
       var brick = SystemAPI.GetComponentRW<Brick>(brickEntity);
```

```
        brick.ValueRW.Hitpoints -= strengthModifier * details.EstimatedImpulse;

        // destroy brick if hitpoints below 0
        if (brick.ValueRO.Hitpoints <= 0)
        {
               ecb.DestroyEntity(brickEntity);
        }
        else
        {
               // update color of the hit brick
               var color =
        SystemAPI.GetComponentRW<URPMaterialPropertyBaseColor>(brickEntity);
               color.ValueRW.Value = math.lerp(config.EmptyHitpointsColor,
        config.FullHitpointsColor,
                      brick.ValueRO.Hitpoints);
        }
}

ecb.Playback(state.EntityManager);
```

Note: Above, we handle the collision events on the main thread, but often you will want to handle them in a job. Unity Physics provides two special job types for processing collision and trigger events: ICollisionEventsJob and ITriggerEventsJob. These job types are demonstrated in several of the Unity Physics package samples.

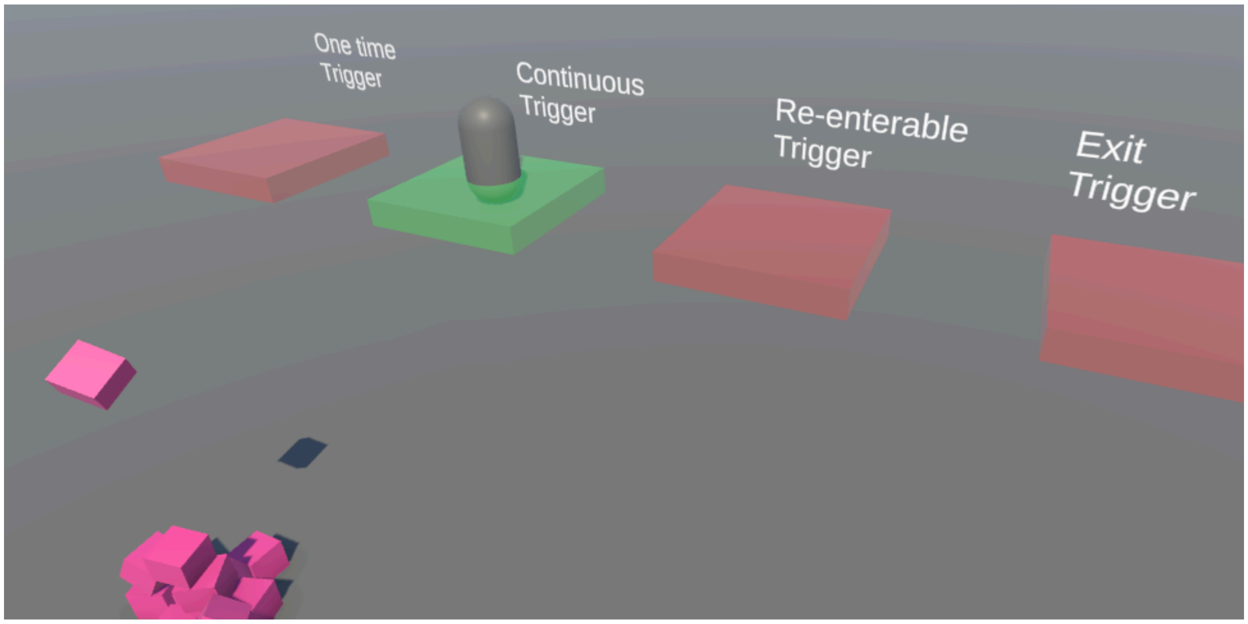
Note: Looping through the collision events can be expensive, so ideally you loop through them only once each update. However, your game logic may need to respond to these events in a multitude of ways, and stuffing many different pieces of logic in one loop can make the code very messy. A good strategy, then, is to generate game-specific events as you loop through the collision events; these game-specific events can then be consumed by other systems that implement your game logic.

Sample: Activation Plate

In this sample, a pink box is spawned when the player walks into a trigger zone. The four different trigger zones have different behavior:

- **One time trigger zone**: Spawns a single box just the first time the player enters the zone.

- **Continuous trigger zone**: Spawns boxes at a certain rate while the player is touching the zone.
- **Re-enterable trigger zone**: Spawns a single box each time the player enters the zone.
- **Exit trigger zone**: Spawns a single box each time the player leaves the zone.



Because Unity Physics is stateless, it has no equivalent of the PhysX “OnEnter” and “OnExit” events, so we must detect enter and exit transitions in our own logic. Therefore, in this sample, the ZoneAuthoring MonoBehaviour gives each trigger zone a Zone IComponentData with a ZoneState enum and ZoneType enum.

ZoneState has four values:

- **Inside**: the player is currently touching the zone
- **Outside**: the player is not currently touching the zone
- **Enter**: the player is currently touching the zone, but wasn’t in the prior update
- **Exit**: the player is not currently touching the zone, but was in the prior update

So as the player moves in and out of a zone, the zone will have Enter and Exit states for the duration of a single update.

ZoneType denotes one of the four spawn behaviours (explained above): **OneTime**, **Continuous**, **Reeneterable**, **OnExit**.

The Zone component also has a LastPhysicsUpdateCount (needed for detecting the Exit state) and a LastTriggerTime (needed for regulating the spawn rate of the continuous trigger type).

The PlayerSystem moves the player based on the user’s input.

The ActivationSystem system does four steps:

1. For the zones that have triggered an event, set the zone's state to either Inside or Enter:

```
Java
// for zones that triggered events this frame, set their state to Inside or
Enter

// get trigger events
var sim = SystemAPI.GetSingleton<SimulationSingleton>().AsSimulation();
sim.FinalJobHandle.Complete();

foreach (var triggerEvent in sim.TriggerEvents)
{
	// determine which body is the player and which is a zone
	Entity playerEntity;
	Entity zoneEntity;

	if (SystemAPI.HasComponent<Player>(triggerEvent.EntityA) &&
		SystemAPI.HasComponent<Zone>(triggerEvent.EntityB))
	{
		playerEntity = triggerEvent.EntityA;
		zoneEntity = triggerEvent.EntityB;
	}
	else if (SystemAPI.HasComponent<Player>(triggerEvent.EntityB) &&
		SystemAPI.HasComponent<Zone>(triggerEvent.EntityA))
	{
		playerEntity = triggerEvent.EntityB;
		zoneEntity = triggerEvent.EntityA;
	}
	else
	{
		// skip because this event is not for the player and a zone
		continue;
	}

	var zone = SystemAPI.GetComponentRW<Zone>(zoneEntity);
	zone.ValueRW.LastPhysicsUpdateCount = physicsUpdateCount; // track when
	the zone was last entered

	if (zone.ValueRO.State == ZoneState.Enter)
	{
		// was Enter, so now should be Inside
		zone.ValueRW.State = ZoneState.Inside;
```

```
        }
        else if (zone.ValueRO.State == ZoneState.Exit ||
                zone.ValueRO.State == ZoneState.Outside)
        {
                // was Exit or Outside, so now should be Enter
                zone.ValueRW.State = ZoneState.Enter;
        }
}
```

2. For the zones that did *not* trigger an event, set the zone's state to either Outside or Exit:

```java
// for zones that did NOT trigger events this update, set their state to Exit
or Outside

foreach (var zone in
        SystemAPI.Query<RefRW<Zone>>())
{
        if (zone.ValueRO.LastPhysicsUpdateCount < physicsUpdateCount)
        {
                // skip because this trigger generated an event this update
                continue;
        }

        if (physicsUpdateCount - zone.ValueRO.LastPhysicsUpdateCount == 1)
        {
                // triggered an event in the prior update but not in this update
                zone.ValueRW.State = ZoneState.Exit;
        }
        else
        {
                zone.ValueRW.State = ZoneState.Outside;
        }
}
```

3. Set the color of entered zones to green and exited zones to red:

```java
// set color of the zones to green when entered, red when exited
```

```
foreach (var (zone, color) in
        SystemAPI.Query<RefRW<Zone>, RefRW<URPMaterialPropertyBaseColor>>())
{
        if (zone.ValueRO.State == ZoneState.Enter)
        {
                color.ValueRW.Value = config.ActiveColor;
        }
        else if (zone.ValueRO.State == ZoneState.Exit)
        {
                color.ValueRW.Value = config.InactiveColor;
        }
}
```

4. Lastly, spawn a box if a zone's conditions are met:

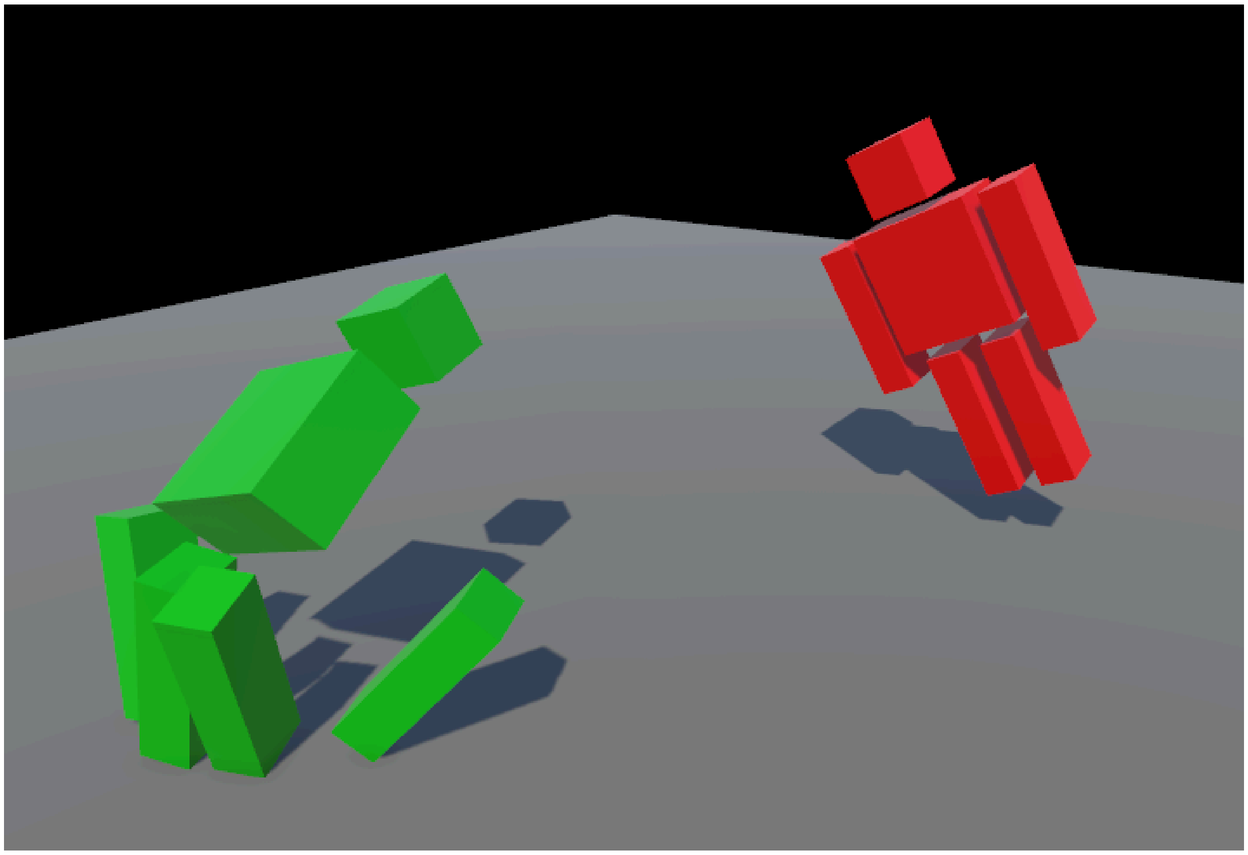
```Java
// possibly spawn a box (depending upon zone state and zone type)

var spawnBox = false;

foreach (var zone in
        SystemAPI.Query<RefRW<Zone>>())
{
        var type = zone.ValueRO.Type;
        var zoneState = zone.ValueRO.State;

        if (type == ZoneType.OneTime && zoneState == ZoneState.Enter)
        {
                // if has not been previously entered
                if (zone.ValueRO.LastTriggerTime == 0)
                {
                        spawnBox = true;
                }
        }
        else if (type == ZoneType.Continuous && zoneState == ZoneState.Inside)
        {
                // if enough time has elapsed since last trigger
                if (elapsedTime - zone.ValueRO.LastTriggerTime >
                config.ContinuousRepetitionInterval)
                {
                        spawnBox = true;
```

```
                    zone.ValueRW.LastTriggerTime = (float)elapsedTime;
                }
            }
            else if (type == ZoneType.Reenterable && zoneState == ZoneState.Enter)
            {
                spawnBox = true;
            }
            else if (type == ZoneType.OnExit && zoneState == ZoneState.Exit)
            {
                spawnBox = true;
            }
    }

    if (spawnBox)
    {
        state.EntityManager.Instantiate(config.SpawnPrefab);
    }
```

Note: For simplicity, the way we track enter and exit events per zone only works for a single object (the player) that might enter and exit. If we need to track enter and exit events for multiple objects, we'd have to do so by tracking these events for each pair of object and zone: either each zone would have a list of objects inside them, or each object would have a list of zones which they are inside. For an example of such a solution, see the "Stateful" events in the Unity Physics package samples.

Sample: Stickman Drop

Stick figure people are dropped on to the ground and break into pieces upon impact.



Each stick figure is formed from separate dynamic bodies connected together by fixed joints. To create the joints, we add the UnityEngine.FixedJoint MonoBehaviour to the head, each arm, and each leg, and we set their “Connected Body” properties to the torso. In baking, this creates a joint entity for each.

For the green stick figure, we set the “Break Force” and “Break Torque” properties of the FixedJoint to very low values, but for the red stick figure, we set these properties to infinity. These values determine the necessary force that will generate *impulse events*. Unlike in PhysX, joints in Unity Physics are not automatically broken apart when the break threshold is exceeded: instead, we must watch for the impulse events and destroy the joints in our own code. In this sample, this is handled by the StickmanSystem:

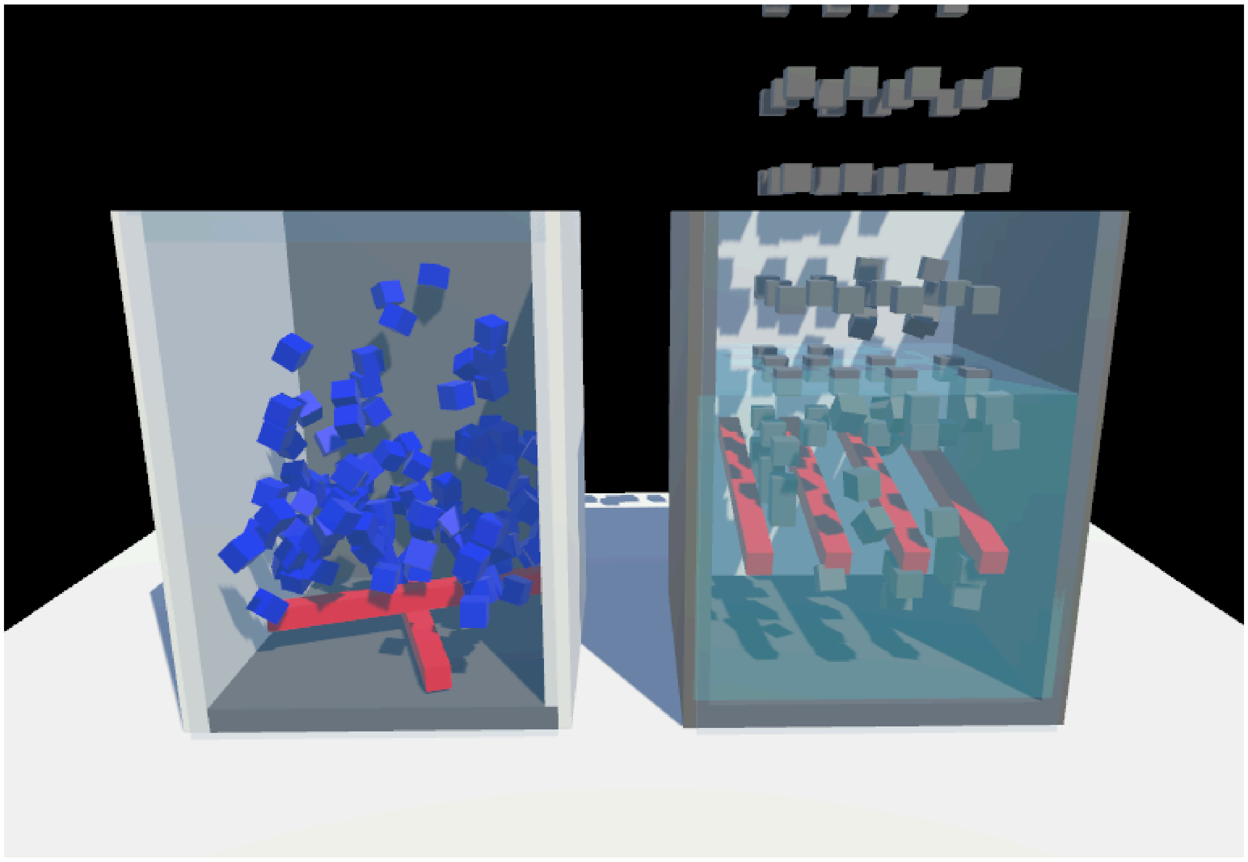
```
Java
// get the impulse events
var sim = SystemAPI.GetSingleton<SimulationSingleton>().AsSimulation();

// to access the impulse events on main thread, we must sync any outstanding
physics sim jobs
sim.FinalJobHandle.Complete();
```

```
var ecb = new EntityCommandBuffer(Allocator.Temp);

// An impulse event is generated when an impulse exceeds a
// joint's Break Force or Break Torque values.

foreach (var impulseEvent in sim.ImpulseEvents)
{
       // An impulse event is generated for both bodies connected by the joint.
       // So we will call DestroyEntity for each joint twice, but this is not a
problem
       // because multiple destroy commands for the same entity in a single ECB
is not an error.
       ecb.DestroyEntity(impulseEvent.JointEntity);
}

ecb.Playback(state.EntityManager);
```

Note: You may notice that the red, unbreakable stick figure is not totally rigid: upon impact, the pieces do not maintain a perfect fixed position relative to each other. This is a limitation of how joints are handled by the solver. To work around this issue, you could instead make the stick figure out of a compound collider rather than separate bodies connected by joints: each time a piece of the body gets broken off, you would have to remove it from the compound collider, and you would also have to make the now separate piece into its own dynamic body.

Sample: Blender

Cubes get hit by rotating blades, and cubes in water (right side) float to the surface.



- The blades are kinematic bodies that are spun by a RotateBladeSystem.
- The water area is a trigger box collider given a BuoyancyZone IComponentData in baking by the BuoyancyZoneAuthoring MonoBehaviour.
- The cubes are dynamic bodies, and the cubes that can float are given a Buoyancy IComponentData in baking by the BuoyancyAuthoring MonoBehaviour.

The Buoyancy component is enableable, and a cube's Buoyancy is enabled by the BuoyancyZoneSystem only while the cube touches the water trigger zone:

```Java
// ...inside the BuoyancyZoneSystem

// disable buoyancy for all cubes
var buoyancyQuery = SystemAPI.QueryBuilder().WithAll<Buoyancy>().Build();
state.EntityManager.SetComponentEnabled<Buoyancy>(buoyancyQuery, false);

// for buoyant cubes inside the buoyancy zone, copy the buoyancy properties and
enabled the buoyancy
{
        // get trigger events
        var sim = SystemAPI.GetSingleton<SimulationSingleton>().AsSimulation();
        sim.FinalJobHandle.Complete();

        foreach (var triggerEvent in sim.TriggerEvents)
        {
               Entity cubeEntity;
               Entity zoneEntity;

               // determine which body is a buoyant cube and which is a zone
               if (SystemAPI.HasComponent<Buoyancy>(triggerEvent.EntityA) &&
                      SystemAPI.HasComponent<BuoyancyZone>(triggerEvent.EntityB))
               {
                      cubeEntity = triggerEvent.EntityA;
                      zoneEntity = triggerEvent.EntityB;
               }
               else if (SystemAPI.HasComponent<Buoyancy>(triggerEvent.EntityB) &&
                      SystemAPI.HasComponent<BuoyancyZone>(triggerEvent.EntityA))
               {
                      cubeEntity = triggerEvent.EntityB;
                      zoneEntity = triggerEvent.EntityA;
               }
               else
               {
                      // skip because this event is not for a cube and a zone
                      continue;
               }

               var zone = SystemAPI.GetComponentRW<BuoyancyZone>(zoneEntity);
               var cubeBuoyancy = SystemAPI.GetComponentRW<Buoyancy>(cubeEntity);

               // copy the zone's Buoyancy data to the cube
               cubeBuoyancy.ValueRW = zone.ValueRO.Buoyancy;

               // enable the cube's Buoyancy so that it will be made to float by
        the BuoyancySystem
               SystemAPI.SetComponentEnabled<Buoyancy>(cubeEntity, true);
        }
    }
```

The cubes with an enabled Buoyancy component are made to float by the BuoyancySystem:

```Java
// ...inside the BuoyancySystem

float deltaTime = SystemAPI.Time.DeltaTime;

foreach (var (buoyant, transform, velocity, mass) in
        SystemAPI.Query<RefRO<Buoyancy>, RefRW<LocalTransform>,
RefRW<PhysicsVelocity>, RefRO<PhysicsMass>>())
{
        float3 currentPos = transform.ValueRW.Position;

        float depth = buoyant.ValueRO.WaterLevel - currentPos.y;
        float buoyancyForce = depth * buoyant.ValueRO.BuoyancyForce;
        velocity.ValueRW.Linear.y += buoyancyForce * deltaTime /
        mass.ValueRO.InverseMass;

        // apply water drag
        velocity.ValueRW.Linear *= 1.0f - buoyant.ValueRO.Drag * deltaTime;
}
```