|  |
| --- |
| Continuing Education Program |
| Physics in Unity |

**1- Why a Physics system?**

|  |
| --- |
| **Copyright Notice**  Copyright © 2013 DigiPen (USA) Corp. and its owners. All rights reserved.  No parts of this publication may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language without the express written permission of DigiPen (USA) Corp., 9931 Willows Road NE, Redmond, WA 98052  **Trademarks**  DigiPen® is a registered trademark of DigiPen (USA) Corp.  All other product names mentioned in this booklet are trademarks or registered trademarks of their respective companies and are hereby acknowledged. |
|  |

A physics system, or more accurately physics engine, is a part of the game engine in charge of updating the behavior of certain objects classified as “physic objects”. Nothing more, nothing else. Generally, the purpose of implementing a physics system within an engine is to replicate “realistic” behavior, although that´s not necessary.

Physics are closely related to game logic, since it´s behavior that is updated in a continuous loop, but physics do not necessarily interact with game object. A common use of physics on triple A games is to provide support for physically believable effects (ragdolls and destructible terrain) that has little to no impact on the gameplay.

Other games rely heavily on physics, and these are called physic based games.

Physics provide a framework to structure gameplay, since physics are nothing more than **a set of general logic rules** that apply to a wide range of objects in the same way. Physic behavior is not coded object by object, but general. Therefore, physics share some characteristics of emergent systems (complex system that spawn from a simple set of rules) and gameplay built on physics has the advantages and disadvantages of emergent systems. Namely, given a workable physic engine, they are fast to implement but hard to control and design.

Physic system, because of their general behavior, tend to be resource intensive (at least more resource intensive that properly coded specific behavior) and therefore physic based games can perform poorly on low end devices unless a lot of care is taken on the implementation and optimization.

**2- Physics in Unity- Introduction:**

Unity does not include its own proprietary physic system. Instead it includes the NVIDIA PhysX engine and provides API calls to access functionality of the engine. However, unless you have access to a source code license, the level of control and access you have to the physics engine is limited to what Unity chooses to make accessible.

If we look closely, the logic behind a physics system can be divided in:

1. **Physical properties updates**:

This takes care of updating the position of the physically enabled objects automatically. This usually means applying linear forces and torques to change the linear and angular velocities AND to update the objects positions according to that velocity.

1. **Collision Detection:**

This takes care of detecting whether a physically enabled object collides with something.

1. **Collision resolution:**

This is related to collision detection, but does not necessarily happen always when collision detection happens. It takes care of resolving the collisions AND adding the specific forces and torques to the physically enabled objects involved in the collisions, so that the next frame they are updated to show realistic or believable behavior.

It´s useful to look at Unity´s integrated physics through these specific lenses, since when we decide if and how our game uses physics, we can also decide which parts of the system we want to use.

**Don´t assume that because you use physics for some type of behavior you need to always use it!**

It is very common, for example to only use the Physical properties update, and resolve and check for collisions manually (especially if all you need are simplified collisions). It’s also common to use the collision detection but ignore the resolution.

The physics system in Unity offers two ways of interacting with it:

-Using the built-in components that the physic system uses to add physical behavior to objects.

-Using the API, both of these components AND the static functions for collision checks.

1. **Components of the Physx system:**

First, we’ll look at the components the physics system uses and we’ll examine them through our three lenses.

When we look at these components, we´ll look at the API, not at the editor exposed variables, since some very important functionality is hidden from the editor.

The main components are:

**-RIGIDBODY**

**-COLLIDER AND PHYSIC MATERIAL**

**-OTHER COMPONENTS**

**-PHYSIC CLASS (STATIC VARIABLES AND MEMEBERS TO CONTROL OVERALL BEHAVIOR)**

**4.1 RIGIDBODY**

Rigidbodies are the fundamental component Unity uses to communicate with the Physx system. A rigidbody component is basically what tells the system an object is physically enabled. **An object without a rigidbody takes no part in the first step of the physics simulation process**. That is, it´s state is not updated by the physics system and thus it’s not simulated.

**4.1.1 RIGIDBODY PHYSICAL UPDATE**

For the PHYSICAL UPDATE step, Unity looks at the following VARIABLES of a rigidbody:

**Position and rotation**: self explanatory.

**MovePosiiton and MoveRotation**: Changes a rigidbody’s position and rotation by APPLYING collision reactions to rigidbodies in contact.

**Velocity**: the distance the rigidbody will move linearly per second. A Vector3.

**Drag**: The amount of velocity the rigidbody loses per second. When used along a constant force (like gravity) high drag will make that force appear less intense (thus simulating “weight”.

**SleepVelocity**: at which velocity the rigidbody goes to sleep. That means it will not be updated until a collision callback is called that applies a force in its resolution, a force is added to the rigidbody OR one of its properties is changed.

**AngularVelocity**:The change of rotation the object experiments over time. This uses world coordinates and therefore should not be set manually. A vector3.

**AngularDrag**: the amount of angular velocity an object loses per second.

**MaxAngularVelocity**: the maximum angular velocity a rigidbody will take. A float. Note that there´s no MaxVelocity.

**Constraints and FreezeRotation**: The constraints to the change of position and rotation of the rigidbody. You can still apply velocities that fall outside of the constraints, but only the non-constrained components will be used. FreezeRotation is a legacy variable before specific axis constraints were introduced.

**CenterOfMass**: the point around which the rigidbody rotates (by using its angular velocity). Its calculated by looking at all the colliders in the object that has the rigidbody attached and its children.

**InertiaTensor and InertiaTensorRotation**: Define the inertia tensor and it´s rotation. The inertia tensor is the distribution of mass along the rigidbody, and defines the tendency to rotate of the rigibody. A force applied in the center of mass will always have only translation and no rotation, but a force applied outside the center of mass will use the inertia tensor to decide how much translation and how much rotation to use. Its calculated by looking at all the colliders in the object that has the rigidbody attached and its children.

**SleepAngularVelocity**: at which angular velocity the rigidbody goes to sleep. That means it will not be updated until a collision callback is called that applies a force in its resolution, a force is added to the rigidbody OR one of its properties is changed.

**Mass**: The mass of the rigidbody. When using forces, more mass means less velocity and angular velocity applied to the object per newton of force.

**UseGravity**: whether the rigidbody is affected by gravity or not. Since gravity is independent of mass, the calculations for it are way simpler than with a standard force, so the physics system provides for an efficient way of simulating it.

**IsKinematic**: Kinematic rigidbodies are not affected by physics, but their physical properties do affect other rigidbodies. They are not moved by collision resolution either.

NOTE: Gravity ignores mass. While setting the mass of a rigidbody wil make it seem heavy when applying a force, gravity is not a force, but an acceleration. Thus it´s drag what will fake the impression of heavy versus light objects.

Also we can modify the PHYSICAL UPDATE of a rigidbody with the following METHODS:

**AddForce**: Add a linear force, thus adds a linear speed relative to the mass.

**AddRelativeForce**: Adds a linear force relative to the rigidbody local coordinate system.

**AddTorque**: adds a torque (rotation force) to the rigidbody. Thus, adds an angular velocity relative to the mass and the inertia tensor.

**AddRelativeTorque**: Adds a torque relative to the rigidbody local coordinate system.

**AdddForceAtPosition**: Adds a force at a position to the rigidbody. Thus it will apply both velocity and angular velocity according to the inertia tensor, the mass and the distance of the point to the center of mass.

**AddExplosionForce**: A shortcut to add explosion type forces (a force that diminishes with the distance to an origin and affects the whole surface of the colliders.

**GetRelativePointVelocity**: Gets the velocity (combined angular and linear) of a pint within a rigidbofy.

**SetDensity**: Sets the mass of a rigiddbody based on the volume of its colliders.

**IsSleeping**: is the rigidbody sleeping?

Sleep: Make the rigidbody sleep.

**WakeUp**: Awake the rigidbody (if it was sleeping).

**4.1.2 RIGIDBODY COLLISON DETECTION**

Rigidbody collision detection is generally taken care of by the engine. However, there are some variables and methods that are related to it.

For the COLLISION DETECTION step, Unity looks at the following VARIABLES of a rigidbody:

**CollisionDetectionMode**: Wheter to use discrete, continuous or continuous dynamic collision modes.

**DetectCollisions**: Whether the collision detection is activate or not for this object.

Also we can check the COLLISION DETECTION of a rigidbody with the following METHODS:

**SweepTest and SweepTestAll**: Returns a collider the rigidbody would collide with if it moved in that direction. Useful for complex rigidbodies with many children colliders. Otherwise RayCast and CapsuleCast (static Physics methods, see below) do the trick.

**OnCollisionEnter, OnCollisionStay and OnCollisionExit (and the Trigger variants)**: Not methods, but messages that are called if the rigidbody has collided OR if a collider has collided with a rigidbody (the messages are shared by the rigidbody and the collider class). No message is called if no rigidbody is present in the collision (since no collision will be detected anyway).

**NOTE 1**: Collision detection modes are worth a look.

First thing to note is that continuous collision detection only works with BOX, SPHERE or CAPSULE colliders.

Consider the following situation:

The ball (which has a rigidbody) moves at the shown speed. That is in frame 1 the ball is on the left of the box and in physics frame 2 the box is on the right.

If the collision detection mode of the ball is discrete, no collision will be detected, since in no frame is the ball overlapping the box.

If collision detection was set to continuous AND THE BOX HAS NO RIGIDBODY, collision detection will be called. However if the box had a rigidbody, it wouldn’t be called.

**For that last case to send a collision, the collision detection mode should be set to Continuous dynamic AND the box collision detection mode is set to either continuous or continuous dynamic.**

This is because how Physx optimizes collision detection. Collider with no rigidbody in the hierarchy are set to a static list for performance reasons. Continuous detection against these static colliders is significantly cheaper than continuous collision detection between dynamic objects, which could get expensive very fast.

NOTE 2: Rigidbodies use all the colliders in a hierarchy to detect mass and pass collision detection to the rigidbody and its components. Thus, **IT’S A BAD IDEA TO HAVE A PARENT AND A CHILD BOTH HAVE RIGIDBODIES**. Only parent should have rigidbodies, never children, especially if a parent higher on the hierarchy has one.

**4.1.3 RIGIDBODY COLLISON RESOLUTION**

The rigidbodies do not take part on the collision resolution (that is done by colliders). The actual response is done in reaction to the forces applied to the rigidbody by the colliders (which call the methods for physical updates). However, two variables can modify collision resolution in a rigidbody.

**solverIterationCount**: How many iterations to go through for each collision, in case the first collision resolution puts the rigidbody into a new collision (colliders call MovePosition and Moverotation in the first step of collision response, and that can generate new collisions)

**useConeFriction**: Whether to use the more accurate and much slower cone friction resolution for the rigidbody. Not necessary unless extreme precision is needed.

**4.1.4 RIGIDBODY DRAWING**

Rigidbodies have an special variable that has nothing to do with the physics system, but is used by the engine to accurately draw the correct position of an object with a rigidbody each Update (that does not necessarily coincide with each Physics update, or FixedUpdate, which is the same).

The variable is the **Interpolation** variable.

Consider the following case. We have a ball with a rigidbody. The blue positions are the positions on the fixed update, of which we have three. Speed is constant. And our current update occurs one third into the THIRD fixed update (T2.33).

T 1 T 2 T 3

P 1 P 2 P 3

Under NO INTERPOLATION, the ball would be drawn on position 2 (P2) since that´s the logical position the ball will have until the next FixedUpdate.

Under INTERPOLATION, the ball would be drawn on position 1 (P1), since it interpolates from the previous to the current position.

Under EXTRAPOLATION the ball will be drawn on position 3 (P3) since it extrapolates the next FixedUpdate position from the object current speed.

Both position and rotation are interpolated.

Note that interpolation drags behind the simulation, but extrapolation can become jerky if sudden velocity changes happen BEFORE the next FixedUpdate (and thus the expected position changes).

Under good coding (no changes of velocity outside of FixedUpdate), there should be no jerkiness under extrapolation and would be the desired setting.

**LAST IMPORTANT NOTE ON RIGIBODIES**:

You should not change the scale of a rigidbody WITH CHILDREN COLLIDERS, since a lot of the physical variables will be recalculated, and performance will change. If you do change the scale, KEEP IT UNIFORM (colliders, except the mesh colliders, do not match non-uniform scales).

**4.2 COLLIDER AND PHYSIC MATERIAL**

Colliders define the “logical volume” of objects for the physics system. That is, only the volume defined by colliders is taken into consideration for any step in the fixed simulation. This helps with the separation of logic from visual representation.

There are four types of collider components, three primitives (BOX, SPHERE and CAPSULE) and a mesh-based collider, but their specific variables are straightforward to understand.

Use of the mesh collider should be restricted, since it´s the slowest. It´s best used for static colliders (with no rigidbody attached. Also, non-convex mesh colliders do not check for collisions between themselves.

**4.1.1 COLLIDER PHYSICAL UPDATE**

Colliders take no part in the physical update of the physics engine (the do send information to that step from the collision resolution, though.

**4.1.2 COLLIDER COLLISION DETECTION**

Collision detection between colliders (provided one of them has a rigidbody somewhere up the hierarchy) is triggered in the colliders overlap (or if the sweep volume of the colliders overlap, if their rigidbody uses continuous collision detection.

For the COLLISION DETECTION step, Unity looks at the following VARIABLES of a Collider

**Bounds**: The world space bounding the collider. This is used to filter collisions before the fine grain pass.

**Enabled**: If not enabled the collision will not be detected.

For the COLLISION DETECTION step, we can use the following METHODS of a Collider

**ClosestPointOnBounds**: to check the closest point in the bounds of the collider.

**Raycast**: Casts a Ray AGAISNT THIS COLLIDER ONLY.

**OnCollisionEnter, OnCollisionStay and OnCollisionExit (and the Trigger variants)**: Not methods, but messages that are called if the rigidbody has collided OR if a collider has collided with a rigidbody (the messages are shared by the rigidbody and the collider class). No message is called if no rigidbody is present in the collision (since no collision will be detected anyway). **EACH COLLIDER TO COLLIDER HIT CAUSES A CALLBACK**. You can have two callbacks called on the same object (if multiple colliders are children of that same object).

To perform the collision detection, Physx uses the static functions of the Physics class that we´ll see below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Collision detection occurs and messages are sent upon collision** | | | | | | |
|  | Static Collider | Rigidbody Collider | Kinematic  Rigidbody Collider | Static  Trigger Collider | Rigidbody  Trigger Collider | Kinematic Rigidbody  Trigger Collider |
| Static Collider |  | Y |  |  |  |  |
| Rigidbody Collider | Y | Y | Y |  |  |  |
| Kinematic Rigidbody Collider |  | Y |  |  |  |  |
| Static Trigger Collider |  |  |  |  |  |  |
| Rigidbody Trigger Collider |  |  |  |  |  |  |
| Kinematic Rigidbody Trigger Collider |  |  |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Trigger messages are sent upon collision** | | | | | | |
|  | Static Collider | Rigidbody Collider | Kinematic  Rigidbody Collider | Static  Trigger Collider | Rigidbody  Trigger Collider | Kinematic Rigidbody  Trigger Collider |
| Static Collider |  |  |  |  | Y | Y |
| Rigidbody Collider |  |  |  | Y | Y | Y |
| Kinematic Rigidbody Collider |  |  |  | Y | Y | Y |
| Static Trigger Collider |  | Y | Y |  | Y | Y |
| Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |
| Kinematic Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |

**4.1.2 COLLIDER COLLISION RESOLUTION**

Collision resolution consist of two steps. The first one is moving the rigidbodies attached to the colliders outside of each other. The amount of movement is relative to the relative mass of a rigidbody with another. If only one rigidbody is present, it will be moved all the way (as if the other rigidbody had infinite mass).

For the COLLISION RESOLUTION step, Unity looks at the following VARIABLES of a Collider

**AttachedRigidbody**: the rigidbody attached to the collider. When adding a collider or changing the parent of a transform, the collider is attached to the rigidbody of the parent transform unless the object attached to the collider has a rigidbody. THIS IS WHY IT’S A BAD IDEA TO HAVE TWO RIGIDBODIES IN A PARENT-CHILD HIERARCHY (you can´t control which gets the callback and which is moved by physics and resolution). If the attached rigidbody is kinematic, there’s no translation resolution applied to it and the other collider (if any) gets sent and infinite mass for translation purposes (as if no rigidbody was present).

**IsTrigger**: A trigger collider send trigger callbacks but DOES NOT PERFORM RESOLUTION on its attached rigidbody or in the rigidbody that collides with it.

**Material and SharedMaterial**: The physics materials of the collider.

**ON PHYSIC MATERIALS:**

Physics materials decide the forces (bounce and friction) applied to the rigidbodies in the point of collision by the colliders. Basically, **they call AddRelativeForce with an intensity based on the bounciness, friction and relative speed**. This has nothing to do with the translation resolution which always happens regardless of physic materials.

This forces are calculated after the translation, thus the speeds used for the calculations are post translation (and thus diminished).

Also, note that kinematic rigidbodies pass their physics material information to whatever rigidbody they are colliding with, even if they ignore the translation step (so there´s friction against them). In a way, we can treat kinematic rigidbodies as rigidbodies with infinite mass, drag and inertia.

This is why if we use MovePosition on a Kinematic rigidbody, any colliding rigidbody will move according to friction (wheter if we just changed the position without calling the rigidbody function, the other object would get the translation resolution but no force or torque applied to it).

**4.3 OTHER COMPONENTS**

Unity has two auxiliary components for the physics system. All they do is add forces each FixedUpdate based on certain parameter.

**ConstantForce**: Adds a constant force every FixedUpdate. Very simple to use.

**Joints**: several classes extend from joint, each with a particular behavior. All a joint does is RESTRICT THE RELATIVE MOVEMENT OF TWO RIGIDBODIES BY ADDING FORCES TO FULLFIL CERTAIN CONDITIONS. What those conditions are and how the forces are added is what defines the different types of joints.

It´s useful to think of joints as links between objects that are simulated and updated by physics (they only take part on the logic update step of the physics engine).

The main joints are:

**SpringJoint**: Simulates a spring tying together two rigidbodies. Restricts separation between them. Rotation is free.

**HingeJoint**: Simulates a hinge tying together two rigidbodies. Restricts rotation between them and fixes the separation.

**FixedJoint**: A fixed union between two rigidbodies. Restricts position and rotation.

There’s also a **ConfigurableJoint** class that can simulate any other kind of joint.

**4.4 PHYSICS CLASS**

On top of the components that Unity provides to access Physx, there are global static variables and methods that we can use to make better use of the engine.

The global variables are:

**BounceThreshold**: Speed at which objects start to bounce (a collision with a speed less than this provides no bouncing).

Gravity: global gravity.

**MaxAngularVelocity,SleepVelocy,SleepAngularVelocity and SolverIteratorCount**: the default variables that can be specifically overwritten by rigidbodies (all rigidbodies use this if not set their own).

**MinPenetrationForPenalty**: the penetration needed for the collision to add a force (on top of the bounce).

**DefaultMaterial**: The default physics material added to every collider if none set.

On top of this, we can access the **Time** class variables **FixedDeltaTime** and **MaxAllowedDeltaTime** to modify the frecuency of the physics updates.

The global methods (many of these are called by the specific collision checks by each collider) are:

**CapsuleCast and CapsuleCastAll**: Casts a capsule in a direction and returns the first collider it finds or all of them.

**CheckCapsule**: Checks collisions against a Capsule (returns only a Boolean)

**SphereCast and SphereCastAll**: Casts a sphere in a direction and returns the first collider it finds or all of them.

**CheckSphere**: Checks collisions against a sphere (returns only a boolean)

**RayCast and RayCastAll**: Cast rays against colliders.

**LineCast**: Casts a line against colliders.

**OverLapSphere**: Like CheckSphere but returns a list of colliders.

**IgnoreLayerCollision**: ignores collisons between two layers.

**GetIgnoreLayerCollision**: return the collision state between two layers (if they ignore or not)

**IgnoreCollision**: Ignores (or stops ignoring) collisions between 2 rigidbodies.

NOTE: The layer to layer collisions (whether they are ignored or not) are set best in the editor and not by script .