The Interloper's Guide To Unity Physics

a.k.a

Unity Physics for the Expeditious

(it rhymes!)

a.k.a

Unity Quick Reference Tables:

Physics Edition

by Nicholas Carter Heinle

http://www.codeartscience.com

Introduction

Reading and watching tutorials, reviewing and modifying other people's well-formed code, and expirimenting extensively on your own are key activities on the path to mastering any new system, but after doing all that for a stretch I personally find it very useful to crystallize, condense, and organize my knowledge and occasionally quickly re-enforce and refresh it in practice.

I've found that quick reference tables make me far more efficient, especially when learning a system for the first time or glancing back at commonly used but slightly-more-complex sets of related system features. I tend to organize concepts in relation to each other and occasionally will print them out and put them on my wall like art -- very functional art, but art nonetheless.

I'd like to share a few of the quick reference tables I've used, this edition being presented applies to the physics engine. Each table also comes with a brief prologue setting up the purpose and context of the table.

Note: These tables are not just for those new to Unity, though newcomers might particularly benefit from re-enforcing and condensing what they learn into core rules, principles and logical arrangements, while not falling into any common -- and particularly insidious -- gotchas. Also, they're great for skilled coders with A.D.D. who just want to get into the code and not scan through every little detail from the disparate and varied sources of Unity information all over the web.

Table 1 Prologue: Rules of Mass and Dimensions

There are some guidelines for setting mass in the physics engine that may seem a bit funky: Unity's physics engine doesn't really care about the "absolute" amount of mass, just the relative amount. Unity also recommends never making the heaviest object in the scene more than 100 times the mass of the lightest. If you break those rules, the physics engine may become unstable.

If that sounds a bit odd/confusing, then you're right, it is not terribly intuitive, but it is a limitation of the calculations involved in the engine. So if you'd like something to be really heavy, make it 10 (kg) and make everything else scaled down accordingly. For simplicity, I often use a mass of "1" for "standard" objects like a human or a small robot, and scale up and down accordingly for larger or smaller objects.

Being perfectly accurate in the absolute sense here is not a necessity for good-looking and well-playing scenarios, though it may be a bit frustrating for those looking for a perfect physical simulation of a ship orbiting a life-size Earth. The key here: play around.

In contrast to mass, you must use the real world height, width, and length in meters of objects to make forces, gravity, and other interactions "look" like they are happening on the right scale (e.g. make your humans generally around 1.8 meters tall). You may apply mass in relative terms but make sure your objects match real world dimensions.

Thus, the Rules of Mass and Dimensions:

MASS and Dimensions:

Rule of Dimensions:	Rule of Mass:
For physics simulations to look "true to scale", the **absolute length**, **width**, and **height** of an object should match those of **real-world objects** in meters. For example, humans should be about 1.8 meters in height, and a typical 4 door-vehicle should be ~4 meters long. If you follow this rule, a giant object falling from the sky observed from a distance, for example, will fall realistically and appear quite massive.	An object's **mass**, on the other hand, is **relative** to other objects in the scene and does not need to be -- and cannot often be -- pegged to absolute real-world values in kilograms. A reasonable range of values is from .1 to 10, though a different range of values may be used. It is recommended that the heaviest object in the scene has a mass no more than 100 times that of the lightest.

Regarding the <u>rule of mass</u>: If this all seems somewhat **ridiculous** -- which, it certainly is if you know any physics or are used to living in our reality -- read the fine print.

WHY DOES THIS RULE EXIST?
THE FINE PRINT:

Simply Put: Computational Limitations

(OK, not such fine print... now for the fine print...) Floating point data types have only so much accuracy, and in order to maintain computational integrity and physical accuracy the range of [values for] mass has to be limited. It's not a Unity problem per-se, it's a problem that you will find in every single physics engine available.

The job of the physics engine is not to simulate reality, which is literally impossible [Editors note: quantum computers, eh?!], but to aid you in creating smoke & mirror techniques that create a believable physics behaviour.

In other words: You [may] have to <u>*fake*</u> it. Choose values that create your desired behaviour, rather than thinking about kilograms and tons.

Thanks to: Chris Herold, at the Unity Forum, for this well stated and detailed "fine print" explanation.

Table 2 Prologue: 5 Simple Ways To Apply Force, 4 Are Recommended

In terms of how to apply forces to objects, Unity gives you many options. You can set a Rigidbody's **velocity vector** manually, to get an instantaneous velocity (you will then experience drag, gravity, and other forces slowing you down over time, from that initial velocity):

```
rigidbody.velocity = new Vector3(10f, 10f, 10f);
```

In Unity, the physics engine updates at fixed intervals X times every second (50 times per second is a common default setting). Each update is considered a "physics frame". During each physics frame a call is made to MonoBehavior's FixedUpdate() method. You can override this method and do all kinds of other physics-related activities:

```
void FixedUpdate() {
        Debug.Log(
                "Physics just updated, " +
                "next time let's do something useful."
        );
}
```

Do not set the velocity vector every physics "frame" as this can lead to problems, in particular, you may override the external calculcations of the physics engine unless very careful. When starting off with Unity Physics, its best to avoid setting the velocity vector alltogether. AddForce() and related methods are an excellent way to accomplish most physics tasks. One example would be applying a constant force, like that from a rocket engine when the engine is "on":

```
void FixedUpdate() {
        if (engineIsOn) {
                rigidbody.AddForce(transform.up * 10f, ForceMode.Force);
        }
}
```

The AddForce() method provides some very useful options beyond applying a standard Newtonian force to an object, as you will see in the force modes table. It offers four overall modes for applying "force" to an object, modes which are broken down into two categories:

1. **Continuous,** and
2. **Instantaneous**

Continuous forces are applied gradually over time, like our rocket engine, whereas **instantaneous** forces are sudden "bursts" of energy, like explosions, balls being fired from cannons, and the like, which instantly add a given amount of velocity to an object.

Note that in each of these two categories, there is one mode that allows you to totally ignore the mass of an object, thus treating objects of any mass identically, and one mode that acts like a real life force, where the heavier an object is, the less a given force will move it.

4 Ways To Apply ***Force***:

Continuous Modes, Applied Each Frame

Force Mode: "Force"

Mass Consideration: **Takes into account the mass of the object.**

Explanation: Applies a given amount of force to an object every physics frame. This will gradually accelerate an object, where higher mass objects will accelerate more slowly than lower mass objects for a given amount of force.

Example: Apply a constant (every physics frame) 1 Newton of force, 1 kg m/s^2: in other words, 1 m/s^2 of acceleration divided by the mass in kilograms:

```
void FixedUpdate() {
    rigidbody.AddForce(Vector3.up * 1f, ForceMode.Force);
}
```

Change the multiplier from 1f to 10f, for example, to apply 10 Newtons upwards instead of 1.

Use Cases: Useful for simulating constant, real physical forces such as a car engine that will accelerate more slowly when more mass is added to the vehicle (e.g. in the form of more cargo or players being added to or thrown off the vehicle as its accelerating).

Force Mode: "Acceleration"

Mass Consideration: **Ignores the mass of the object.**

Explanation: Applies acceleration every physics frame, accelerating at m/s^2, ignoring the amount of mass completely and thus giving "simplified" control over an object’s acceleration/speed.

Example: Apply 1m/s^2 of acceleration constantly, regardless of mass (every physics frame):

```
void FixedUpdate() {
    rigidbody.AddForce(Vector3.up * 1f, ForceMode.Acceleration);
}
```

Change the multiplier from 1f to 25f, for example, to apply 25 meters per second per second of acceleration.

Use Cases: This is a great "cheat" for a getting a variety of objects possessing different masses to move in a similar fashion, like a swarm of varied bots or a fleet of flying ally ships that may differ in mass, dimension, and shape but should move at similar speeds and accelerations. Just find an acceleration rate that works well in one case while testing, and it will work well for all of them. Useful for coordinating group flight paths and movement patterns.

Instantaneous Modes, Applied Once

Force Mode:	"Impulse"
Mass Consideration:	**Takes into account the mass of the object.**
Explanation	A one time shot of velocity, basically "how much faster" do you want this object to go in m/s, divded by kilograms of mass.
Example	Without need for gradual acceleration, instantly increase the velocity by 1 m/s, divided by the total mass in kilograms: `rigidbody.AddForce(Vector3.up * `**`1f,`**` ForceMode.Impulse);` At the end of this method call assuming no previous velocity and a weight of 1kg, the rigidbody will be travelling upwards at 1 m/s. Change the multiplier from 1f to 14f, for example, to apply 14 m/s of velocity (divided by total mass in kilograms).
Use Cases	Getting hit by an explosion, receiving a blast from an energy weapon, a ball being shot from a cannon, all cases where the change in velocity is basically instantaneous, and where the velocity lessens as the mass of the propelled object increases.

Force Mode:	"Velocity Change"
Mass Consideration:	**Ignores the mass of the object.**
Explanation	A one time shot of velocity, where you can simply say "how much faster" you want to go in m/s, ignoring any mass considerations. This is similar to setting the velocity vector directly as mentioned above, although it adds to the existing vector.
Example	Without need for gradual acceleration, instantly increase the velocity by 1 m/s: `rigidbody.AddForce(Vector3.up * 1f, ForceMode.VelocityChange);` At the end of this method call assuming no previous velocity, the rigidbody will be travelling upwards at 1 m/s. Change the multiplier from 1f to 16, for example, to apply 16 m/s of velocity.
Use Cases	If you want to "cheat reality" and have something suddenly moving at a given speed **regardless** of mass, this can be of great value. In other words, a cannon that fires anything out of it at the same speed no matter the mass of the projectile.

Table 3 Prologue: The Yin and Yang of Movement

In Unity, there are generally two ways to translate (i.e. move) and rotate an object in 2d or 3d space:

1. by manipulating its Transform -- with help from a Tween, for example -- or
2. by utilizing the physics engine via its Rigidbody and applying forces, etc.

It is very important to understand the fundamental nature of these two (often) very different ways of handling movement. In practice, the two are best combined to handle a variety of circumstances powerfully and efficiently. Thus, the Ying and Yang of movement:

The Yin and... ...Yang (of Movement)

The Way of the Transform

At Its Core:
Absolute Control and Simplicity.

How it works:
Modify a **transform** incremenetally over time, moving and rotating an object through space, calculating and assigning its exact positions at each point along its path. Often this is done indirectly through use of a tween.

Example:

```
Tween.
  MoveTo(
    gameObject, // what
    new Vector3(1,1,1), // to where
    2f // in how much time
  );
```

Pros: Gives you total control over where an object starts, ends, and what speed it travels during that time as well as the overall "shape" of the movement (linear, curved, etc). Tweens libraries like iTween, HOTween, and LeanTween make this very simple and direct.

Cons: In terms of movement, tweens or your own movement calculations are obviously not a detailed physics simulation (unless you really go all out with your logic) and thus you miss out on the many wonderful and subtle details of a physical simulation as well as automatic collision detection and reactions. Tweens can "hand off" to a physics simulation though. For example, when getting to the end of a tween movement, enable physics, copy over the velocity to the rigidbody, and allow for a realistic collision with a large number of objects.

The Way of the Rigidbody

At Its Core:
Complex Emergent Interactions.

How it works:
Apply forces or modify the translational and angular velocity of an object via its **Rigidbody** and let the physics engine do the rest of the work, including collisions, bounce, drag, friction, gravity, angular momentum, et al.

Example:

```
rigidbody. // what
  AddForce(
    Vector3.up * // direction
    1f, // quantity of force
    ForceMode.Impulse // force mode
  );
```

Pros: Gives you a beautiful and realistic physics simulation with all the details relating to forces, drag, friction, bounciness, momentum, mass, gravity, and complex collisions. Great when you're willing to expiriment with different values and settings and see what works best (or calculate what works best in some cases).

Cons: Not always the best method for artfully getting an object from point A to point B, for example. Movement can be hard to control precisely due to emergent behaviors and the many variables involved. Again, you may have to experiment, carefully calculate, or occasionally override parts of simulations with a tween or manual position updates to get your specific desired behavior. Physics simulations can be more computationally intensive as well due to the larger number of variables when compared to a more succint tween.

Table 4 Prologue: 5 Simple Guidelines for the Everyday Rigidbody

This brings us to some basic guidelines for the component that is most important in achieving these goals "the rigidbody way" (and possibly in concert with "the transform way"): the Rigidbody.

5 Simple Guidelines for the Everyday RIGIDBODY:

1. Moving Colliders ♥ Rigidbodies

Its more than a love, its a **need**. Any GameObject that moves and has a Collider should have a Rigidbody on its GameObject or a parent. **Even if a GameObject doesn't use the physics engine explicitly and uses the transform for movement at all times, it still needs a physics-disabled (isKinematic property = true) rigidbody**. See the fine print for details as to why this is a must not a should.

2. You Can Disable Physics At Any Time

If you don't want your GameObject to obey the laws of physics once you've added a Rigidbody -- say, you want to do things the "non physics way" -- you can "disable physics" for the GameObject by setting the Rigidbody's **isKinematic** property to **true**. When you disable physics for an object it will freeze in place unless moved directly and explicitly via its transform. It will now "ignore" physics interactions, with one exception, because of guideline #3...

3. There Are Many Levels of "Disabled Physics"

...quite notably, your "physics disabled" GameObject **will still push around and displace** physics-enabled GameObjects it collides with all the while ignoring "incoming" collisions itself and otherwise being totally unaffected by physics interactions while **isKinematic = true**.

If you want to do away with the "pushing around other physics-enabled objects" behavior, a simple, powerful technique is to set the Collider **isTrigger** property to true, which changes the nature of the collider, causing it to -- among other things -- ignore those outgoing collisions.

Any object in this state will pass right through all other objects while still receiving object enter/stay/exit events in the form of OnTrigger events, instead of OnCollision events.

4. And... You Can Re-Enable Physics at Any Time...

The **isKinematic** property can be set to false **at any time** to suddenly "re-enable physics" on your GameObject. This is very useful at the end of a tween animation during some meaningful collisions with other objects, or during an explosion when you want lovely explosive effects and

5. But Do Not Nest Your Physics-Enabled Rigidbodies... Period

Don't make a physics-enabled rigidbody the child of another physics-enabled Rigidbody in the GameObject heirarchy, or you are likely to experience undesirable behavior. Always use Fixed joints, Spring joints, or physical joints of some kind to connect separate rigidbodies, do not leave it to the

behavior, among many other scenarios. The ability to enable/disable physics at any time is also a nice way to implement a pause system or freeze time for some objects but not others, because when you disable physics for an object it will freeze in place unless moved directly and explicitly via its transform. When pausing this way, you can save the velocity vector the moment before you disable physics, and re-assign that value when re-enabling physics to cause an object to continue moving on its previous path (assuming you didn't modify its position). If you're familiar with Time.timeScale, realize that setting Time.timeScale to 0 may be overkill for a pause technique in many scenarios.	parent-child heirarchy to provide for linked movement. This rule does not apply to a child Rigidbody with physics **disabled**, however. Thus if you must add your physics-enabled rigidbody as a child to another at some point in a scene, you can simply disable physics by setting the child Rigidbody's **isKinematic** property to **true** before adding it, then add its mass to the parent manually. The parent Rigidbody will then warmly embrace the child into its physical form, colliders and all, as if joined by an unbreakable, unstretchable, and unconditional bond (the kind of parent-child bond we all crave at some level, but notably, a topic for a very different kind of table... we digress).

Regarding the <u>1st guideline</u>: Why does any moving object with a Collider need a Rigidbody? Let's go into a bit of detail.

WHY DOES THIS GUIDELINE EXIST?
THE FINE PRINT:

Simply Put: Collision Optimization

You can indeed have a GameObject with a Collider and no Rigidbody. If there's no Rigidbody then Unity assumes the object is static, non-moving. Unity does not bother testing for collisions BETWEEN static objects. As you can imagine, this is very efficient if you have lots of scenery the player can bump into.

So the purpose of having a kinematic Rigidbody (a Rigidbody with isKinematic set to true), rather than no Rigidbody, is to turn on collision detection between this object and all other Colliders in the scene (even the static ones.) Effectively you are letting Unity know that this object moves around, so Unity will then do collision-detection between it and everything else.

[Editor's note: **The Big Warning:** if Unity thinks the Collider is static, and then you move it, the "scene graph" -- optimized for collision detection with static objects -- will be recalculated, which will slow things down substantially in large setups. Thus, if you plan to move something, and it has a Collider, add a Rigidbody. For even more information about this, see: http://docs.unity3d.com/Documentation/Manual/Physics.html and take your time playing around.]

Thanks to: Matt Diamond, a.k.a. Bampf, at UnityAnswers for this well stated and detailed "fine print" explanation.

Appendix: Using Drag Correctly: A Quick Review

Let’s review a few scenarios to get a better idea of how drag and mass affect the behavior of an object with regard to physics:

- Having a relatively high mass (try 10) will make an object push lower mass objects (try 1) around and cause the higher mass object to react very little to those lower mass objects. But increasing mass will not cause an object to fall faster: as you learned in your high school physics class, only **decreasing** the **drag** will increase the speed of an object’s descent.
- Thus, with high drag you can have a giant, heavy feather (mass 10, drag 10) that falls slowly but knocks everything aside wildly when it hits lower mass, lower drag objects (say, mass .1, drag 0.1). To prevent those little objects from moving wildly, set their drag way up to 10, and they will be moved forcefully upon collision with the giant feather, but quickly come to rest because of their drag. You can add friction against various surface materials to this situation as well to make things even more interesting by adding Physic materials to your colliders.
- As we noted, increasing an object’s drag to a high value will make it fall like feathers or Styrofoam (again, try a value of 10), conversely decreasing the drag to near zero will make it fall like a gold brick (try 0.1). The lower the drag, the higher the apparent "density" of the object. Well, not quite:
- In the real world, the amount of exposed surface area in the direction of an object’s movement and other aerodynamic properties of that object -- relative to its mass -- are primarily what determine the speed of an object falling, but Unity let’s us **imply a fixed exposed surface area (more accurately, aerodynamic inefficiency) to mass ratio** via the drag property. Technically, if you wanted to be more accurate than a simple "fixed ratio", you could change the drag property dynamically, reducing it when less surface area is exposed to the direction of movement, and increasing it when more is exposed. Since the geometric properties of the surface area and object are important to its aerodynamic inefficiency, you could take this yet further and do more detailed air flow calculations as well to determine more accurate aerodynamic inefficiency values, but for most games this is not worth the trouble, and could be intensive computationally (just try calculating turbulence, hah!).
- You can also use high drag in various slightly "hacky" ways: to make an object "easy to control" in a cartoonish flying game, such as with a hovering, rocket-powered flying vehicle that you want to quickly stabilize when you’re not applying turns or forces to it. High drag makes flying it far easier for the user because its not jumpy and will stop turning and moving rather soon after they stop pressing buttons. You could also do away with drag and write stabilizers yourself of course.