The scale between Unreal world and Karma physical world

By Jijun Wang

This report is basically a response to Marco’s Mass and Gravity test report (http://digilander.libero.it/windflow/eng/reports.htm). I was attracted by Macro’s experiments/reports and started to think about these important problems that were ignored for a long time. There were some issues I got confused from the reports. So following Marco’s approach, I repeated and did some extra experiments to help me clear the confused issues. This report summaries what I have learned form Marco’s report and what I found from my experiments. The ultimate goal is trying to find out the scale used in Unreal Engine when it interacts with Karma physical engine. Based on this scale, we can have our own correctly scaled world.

In this report, I attempt to describe the scale used in Unreal Engine in a systematic and clear way. At first, I would like to copy and print Karma User Guide’s explanation about “Units and Scaling” here:

> “**Units and Scaling**
>
> There is no built-in system of units in Karma, which is not to say that quantities are dimensionless. Any system of units may be chosen, either meter-kilogram-seconds, centimeter-grams-seconds or foot-pound-seconds. However, the programmer is responsible for the consistency of values and dimensions used.
>
> The range of masses and lengths within which Karma will perform well is limited by the precision of the underlying floating point implementation: single precision floating point (the default build option for Karma on all platforms) provides for approximately six significant decimal digits. In order to support a variety of scales of length and mass, Karma requires the application to supply default values of mass and length.
>
> For example, if a human is a medium-sized object in your application, with height 1000 units, and mass 4 units, 1000 is a reasonable value for default length, and 4 a reasonable value for default mass. Input masses and lengths should not be enormously different to the defaults: about two orders of magnitude either way is reasonable in single precision. Karma scales all its internal tolerance and threshold values in accordance with the values you supply, but if you override any of these defaults, you too need to scale your values appropriately. Time is assumed to be in seconds, and angles measured in radians. If your time and angle units are different from this Karma’s defaults will not be appropriate, and you will need to rescale some of the default values yourself in order to get the best behavior from Karma. (The units for the default parameters to the world are given in appendix A)
>
> If, for example, you are working in degrees for angular measurement, you will need to scale Karma’s default angular velocity and acceleration thresholds for auto-disabling by 180/pi. And if the time system with which you wish to use Karma advances the simulation by 1 unit of time for 1/60s, you should scale all Karma’s thresholds for auto-disabling: velocities by 1/20, and accelerations by 1/3600."
>
> --Karma User Guide 1.0

From above, we know that time, length, and mass are the three basic unit scales Unreal Engine needed to input to Karma engine. I will evaluate them one by one. And then evaluate Force and Torque.

Time scale

Unreal engine claims that it uses second to measure time. So we all think the time scale is 1. That is:

1 UU time = 1 second in real life

Unfortunately, we are all misled. Every game has its own time system that is called GameSpeed. By default, the GameSpeed is set to 1 in UT2003/4. This means that 1 second in the game equals to 1 second in the computer system. We can use command ***SLOMO float*** to set the GameSpeed to any number to fast or slow the game. For example, *SLOMO 2* will fast the game by making 2 second in the game is 1 second in the computer system. Considering Karma engine was NOT added to Unreal engine from the beginning, I doubt Karma could be integrated into Unreal so deep that they use totally the same time system. If they use different time systems, a time scale should exist! If for default GameSpeed=1, there is a time scale S_t, then for other GameSpeeds, the time scale will be S_t*GameSpeed.

Marco reported that **g** (acceleration of gravity.) was set by Gravity in PhysicsVolume. By default, g = 950. No matter what unit scale is used between Unreal and Karma, with g = 950, we should get the same value from the gravity test experiment since the setting and testing are all done on Unreal side. In Marco's testing, he got g = $2*8192/4.58^2 = 781$. I also got similar results. So what's wrong? We are using the game time to measure **g**. If Karma uses different time system, then this will cause the errors.

Let's assume Karma using the computer time system. We modify the Gravity test experiment to print out the computer system time[1], game time, and Bullet's location. Then with these data we can draw the time-location curve and with 2-order data regression we can measure the acceleration. If Karma utilizes Newton's law to a falling object, we should have

$S = ½ g (t-t_0)^2 = ½ g t^2 – gt_0t + ½ gt_0^2$ *where t_0 is the time we begin to observe the falling activity.*

Figure 1 shows the time-location curve measured by computer system time. The 2-order regression gives us a perfect match (R^2=1). This shows that Newton's law is working and the measured g = 471.07*2 = 943 is very close to the setting value 950! Comparing the computer system time and the game time, for my computer system, I got

1.1 game time = 1 second computer time

If the **g** error between game time and computer time is caused by the time scale, then we should have $1.1^2 * g_{game} = g_{computer}$. To verify this, let's calculate:

$1.1^2 * g_{game} = 771*1.1^2 = 933$ which is close to $g_{computer} = 943$

To further observe the time scale effect, let's set GameSpeed=2 and do the gravity testing again. In Figure 2, we got g = 3732 in computer time which is close to $950*2^2$ =

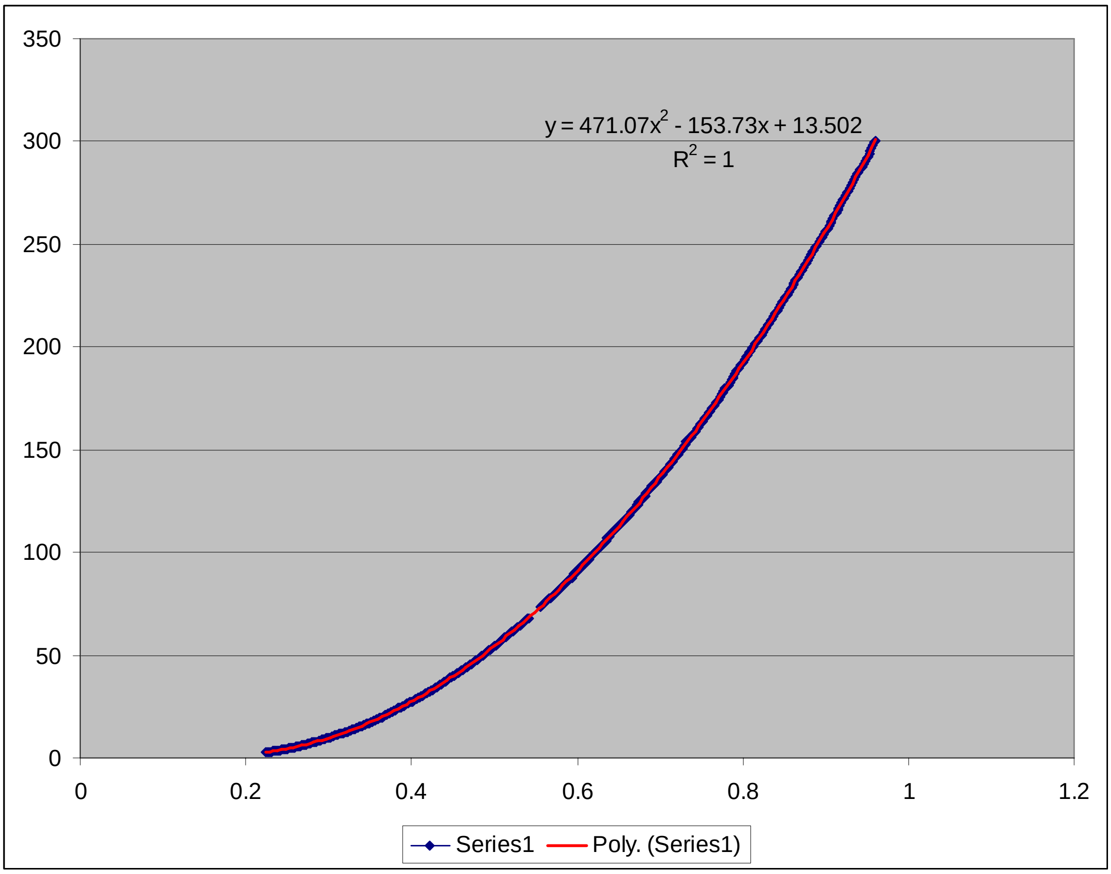
[1] StopWatch function was used in the experiment to print out the computer system time. Please note that there is a known data wrap bug in StopWatch which may print out negative time values when the measured time is in the order of second. However for the positive values, StopWatch works well. So in all the experiments, only positive time was used in measurement.

3800. And in game time, we still got g = 771. Now we multiple 771 with the square of 2*1.1, we will get 771* 2.2^2 = 3732 which is the g measured in computer time.

In Figure 3 and 4, I did two more experiments where g was set to 1900. And I still got the same result.

Therefore, in summary, in Unreal world and Karma world, different time systems are used. For a GameSpeed, in my computer system[2], we have

1.1* GameSpeed UU time = 1 second

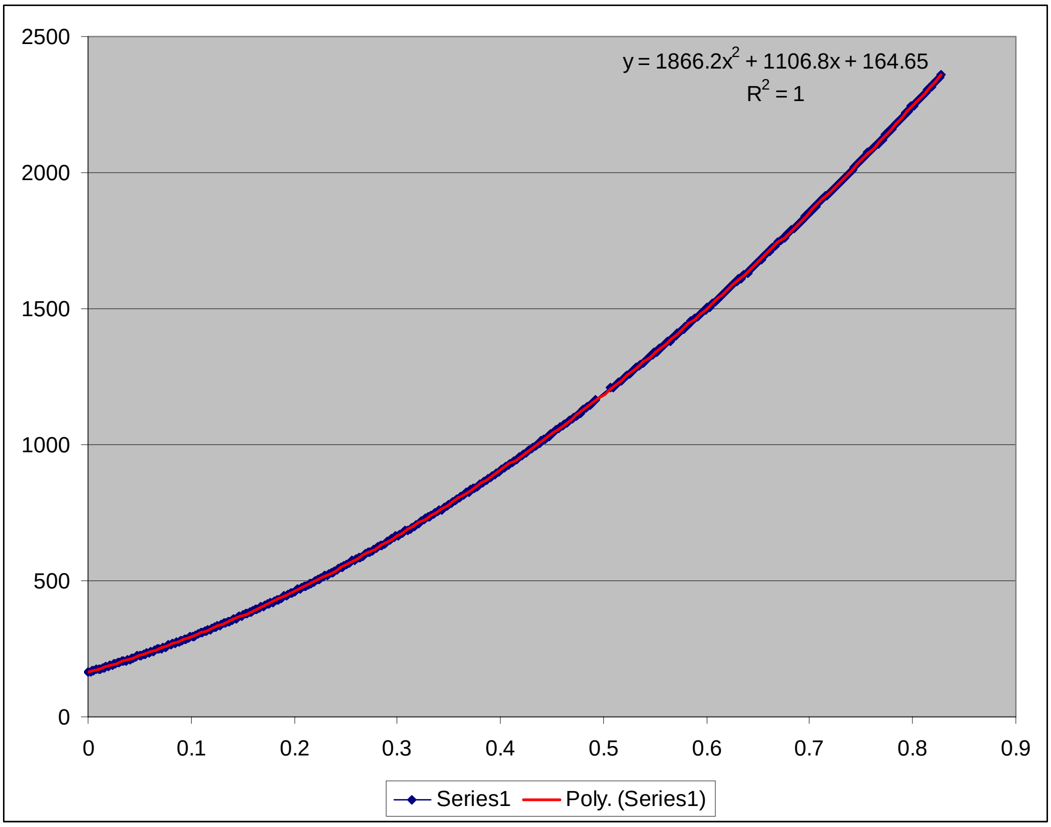


[2] The constant 1.1 may change from one computer system to another because of the different capabilities

In computer system time:
$g_{setting}$ =-950 UU/s^2
$g_{observed}$ = 471.07*2 = 943 UU/s^2
GameSpeed = 1.0

In game time:
$TotalFallingTime_{level}$ = 4.61 s
$g_{observed}$ = 2*8192/t^2 = 771 UU/s^2

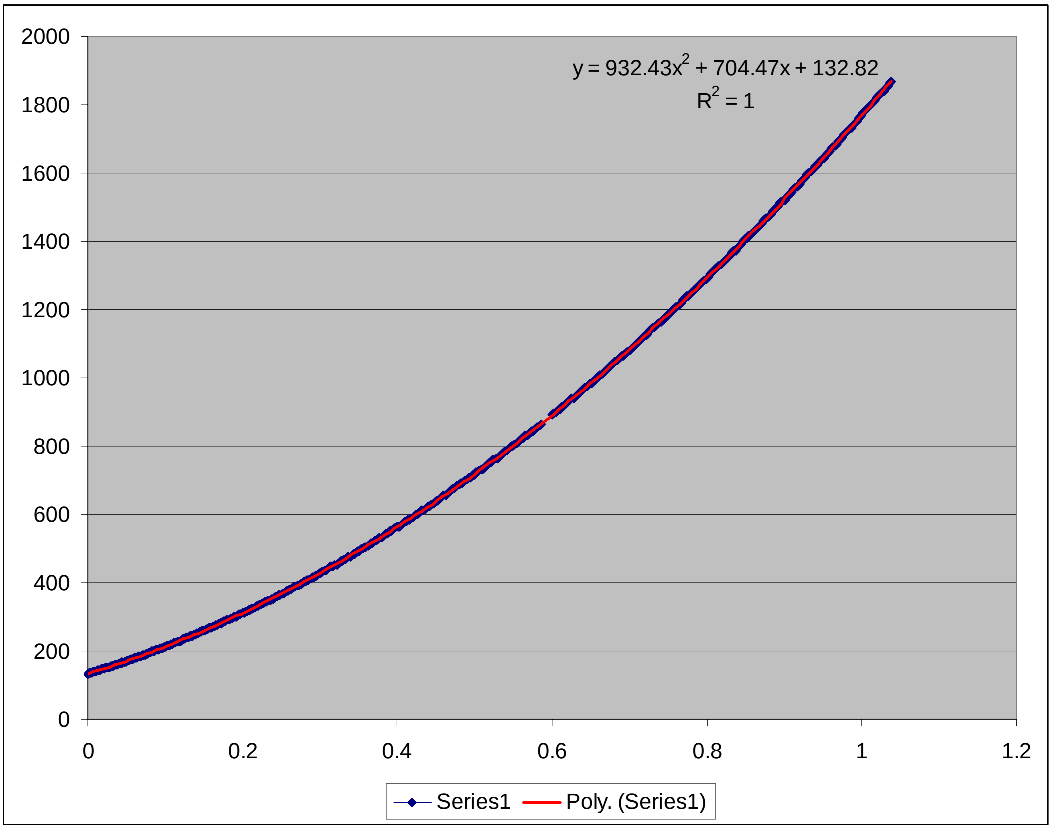
Figure 1 Gravity Testing 1



In computer system time:
$g_{setting}$ =-950 UU/s^2
$g_{observed}$ = 1866.2*2 = 3732 UU/s^2
GameSpeed = 2.0
$g_{setting*}$ GameSpeed^2 = -3800

In game time:
$TotalFallingTime_{level}$ = 4.61 s
$g_{observed}$ = 2*8192/t^2 = 771 UU/s^2

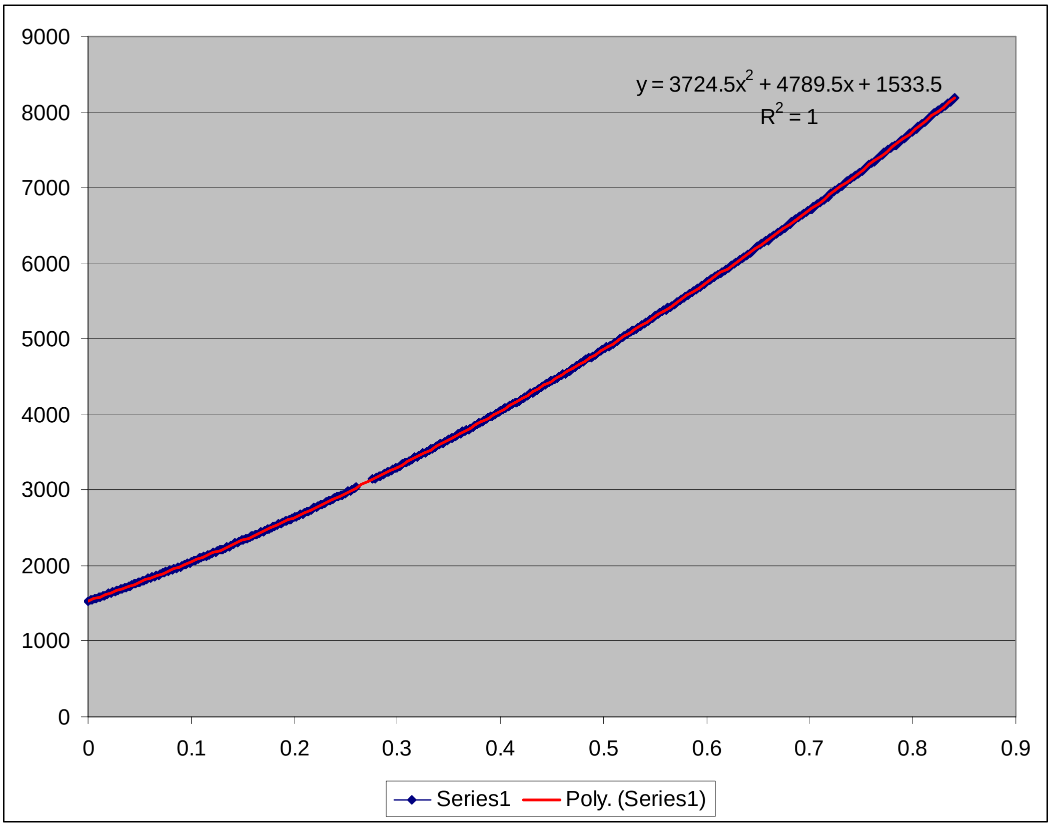
Figure 2 Gravity testing 2 with GamseSpeed=2



In computer system time:
$g_{setting}$ =-1900 UU/s^2
$g_{observed}$ = 932.43*2 = 1865 UU/s^2
GameSpeed = 1.0

In game time:
$TotalFallingTime_{level}$ = 3.26 second (in game)
$g_{observed}$ = 2*8192/t^2 = 1541.6 UU/s^2

Figure 3 Gravity test 3 with setting g=-1900



In computer system time:

g_{setting} =-1900 UU/s^2

g_{observed} = 3724.5*2 = 7449 UU/s^2

GameSpeed = 2.0

$g_{\text{setting*}}$ GameSpeed^2 = -7600 =roughly= g_{observed}

In game time:

$\text{TotalFallingTime}_{\text{level}}$ = 3.26 s

g_{observed} = 2*8192/t^2 = 1541.6 UU/s^2

Figure 4 Gravity test 4 with setting g=-1900, GameSpeed=2

Length scale

This is the easiest part. In UDN (http://udn.epicgames.com/Two/KarmaReference), we are told that

50 UU = 1 Karma UU

If we assume meter-kilogram-second system, then we have 50 UU = 1 meter which is roughly consistent with the Unit rule in UnrealWiki (http://wiki.beyondunreal.com/wiki/Unreal_Unit) that 52.5 UU = 1 meter. Again, considering Karma was added after Unreal engine already exist, I think this minor inconsistence is reasonable[3]. So I believe

[3] It's also possible that foot-pound-seconds system is used. But considering 50UU=1KUU and 52.5UU=1meter, I believe meter-kilogram-second system is more reasonable. If foot-pound-seconds system is used, since 16 UU = 1 foot, we should be told 16 UU = 1KUU.

50 UU = 1 meter

The last question we need to answer is that why by default g = 950 UU/s^2 = 19 m/s^2 (if we hold 50 UU = 1 meter)? From what I learned from Karma User Guide, there is no default constant **g** in Karma world (Ok, if we must say there is a default value, then from the manual's Appendix A, it's ZERO). **g** is supposed to be set by the user (the developer). In Unreal engine, it's the parameter Gravity in PhysicsVolume. So there is no sufficient reason to assume **g** must be (or close to) 9.8 and use it as reference to deduct the length scale even this is an obvious common sensor we should hold. Or let me say it in another way, we needn't limit ourselves to the number 9.8 and the default value 950. We can just set it to any number we think it should be if all the other unit scales fit well with the physical world. If I must explain why it's 950 not other number, then my guess (usually this means I'm wrong ☹) is that the value was picked up to have the final whole feeling to be close to the real world. This may include considering the time scale problem, considering the air/wind effect (KLinearDamping, KAngularDamping …) etc…

Mass scale

Since F = m a, let's consider mass and force together. To better understand the mass effect, I modified Marco's experiment a little bit. I put the experiment world in a PhysicsVolume with Gravity = (0,0,0). If no force applies on the object, it should stay in the air steadily. The experiment result shows that this is true. Now, if we apply a force on the object, it should fall in the acceleration a = F/m. With the measured a, we can figure out the scale used in mass and force.

In Figure 5, an object with mass=1.0, kMass=100 was push down by force F=100 and measured in computer system time. The testing result shows that

a = 25.988*2 = 52 UU/s^2 = 1.04 m/s^2

With the F and m, we can get a = 100 UU_{force} / 100 UU_{mass}. So we have

1 UU_{force} / UU_{mass} = 50 UU_{length}/s^2 = 1 m/s^2

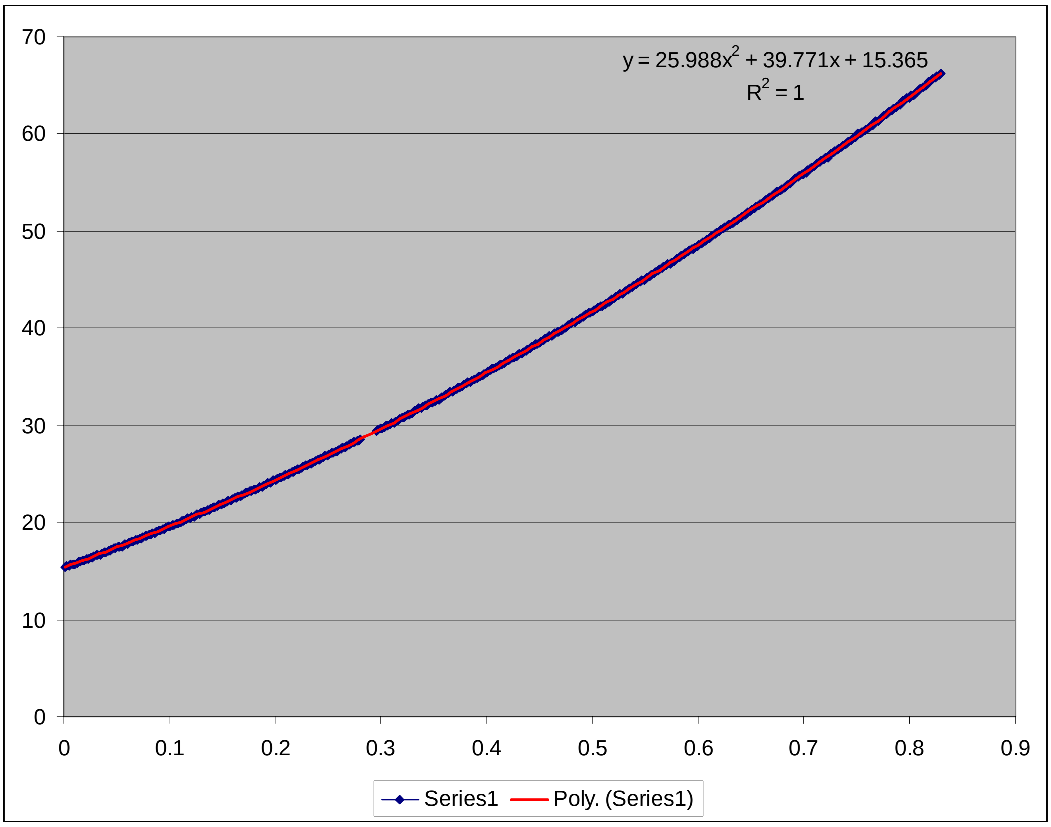
if we assume

1 UU mass = 1 Kg

then

1 UU force = 1 N

To confirm this relationship, I conducted another experiment with F=200. In Figure 6, we can see that the measured a = 103 UU/s^2 = 2.06 m/s^2 is close to a = F/m = 200/100 = 2 m/s^2.



$g_{setting}$ = 0 UU/s^2

mass = 1.0

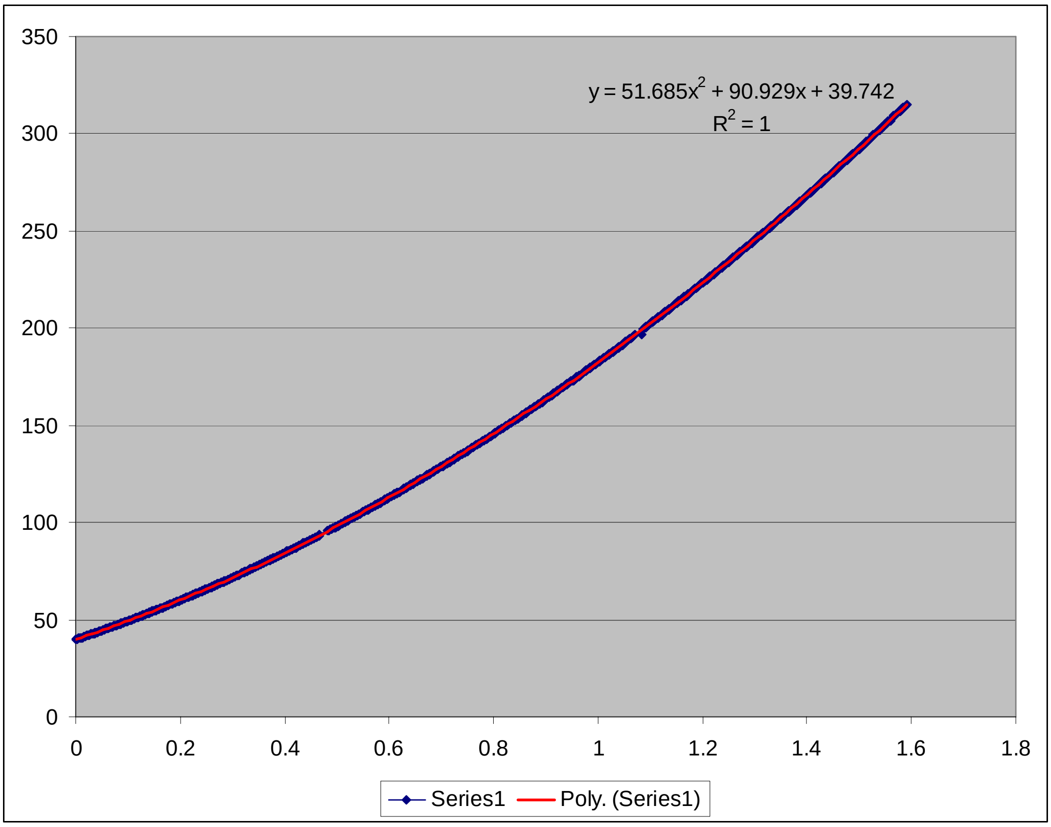
kMass = 100.0

DrawScale = 0.6 StaticMesh=brick

F = 100 UU

a = 25.988*2 = 52 UU/s^2

Figure 5 Force testing 1 with F = 100 UU



$g_{setting}$ = 0 UU/s^2	DrawScale = 0.6 StaticMesh=brick
mass = 1.0	F = 200 UU
kMass = 100.0	a = 51.685*2 = 103 UU/s^2

Figure 6 Force testing 2 with F = 200 UU

However there is one problem. If Unreal uses mass-length-time system, from F=ma, we should get 1 UU_{force} = 1 UU_{mass} * 1 UU_{length}/s^2 = 1/50 UU_{mass} * m/s^2. It seems that Unreal holds 50 UU_{length} = 1 meter, but for mass and force, it uses a simple 1:1 scale! To proof this, let's look at torque.

To find out the scale used in torque, I designed the following experiment shown in Figure 7. The board and the box were put in a non-gravity physics volume. We apply two forces on the box, one pushes the box down and one pulls the box to leave the board's center (the hinge). To keep the board being steady, we need to apply a dynamic torque to balance the push force and this torque should change as the box moving on the board. Once the box totally leaves the board, the applied torque should turn the board over…

In the first attempt, the applied torque is T = F*Distance(box,board) in UU unit. The board totally can't balance. Then I changed to T = F*Distance(box,board)/50. Now the physical world behaved perfect: the box moved on the board as if on a horizon floor (The board's rotation angle is showed in Figure 8). So from the experiment, we get

1 UU_{torque} = 1 Nm

Again, if we assume Unreal use mass-length-time system, we should get 1 UU_{torque} = 1 UU_{force} * 1 UU_{length} = 1/50 UU_{force} * meter which is inconsistent with our observation. So

I believe Unreal uses 50:1 scale in geometry system and 1:1 scale in dynamic system. That is:

50	**UU length**	=	**1**	**meter**
1	**UU mass**	=	**1**	**Kg**
1	**UU force**	=	**1**	**N**
1	**UU torque**	=	**1**	**Nm**

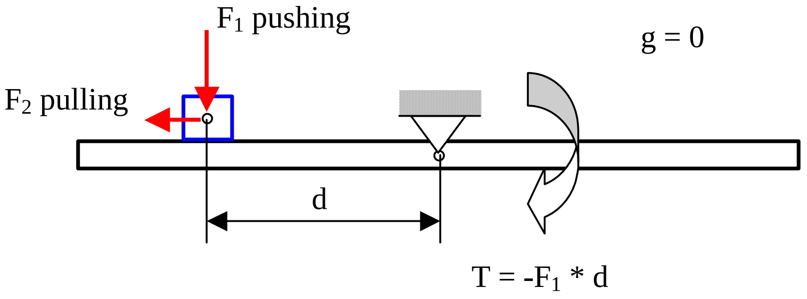


Figure 7 The torque balance experiment design

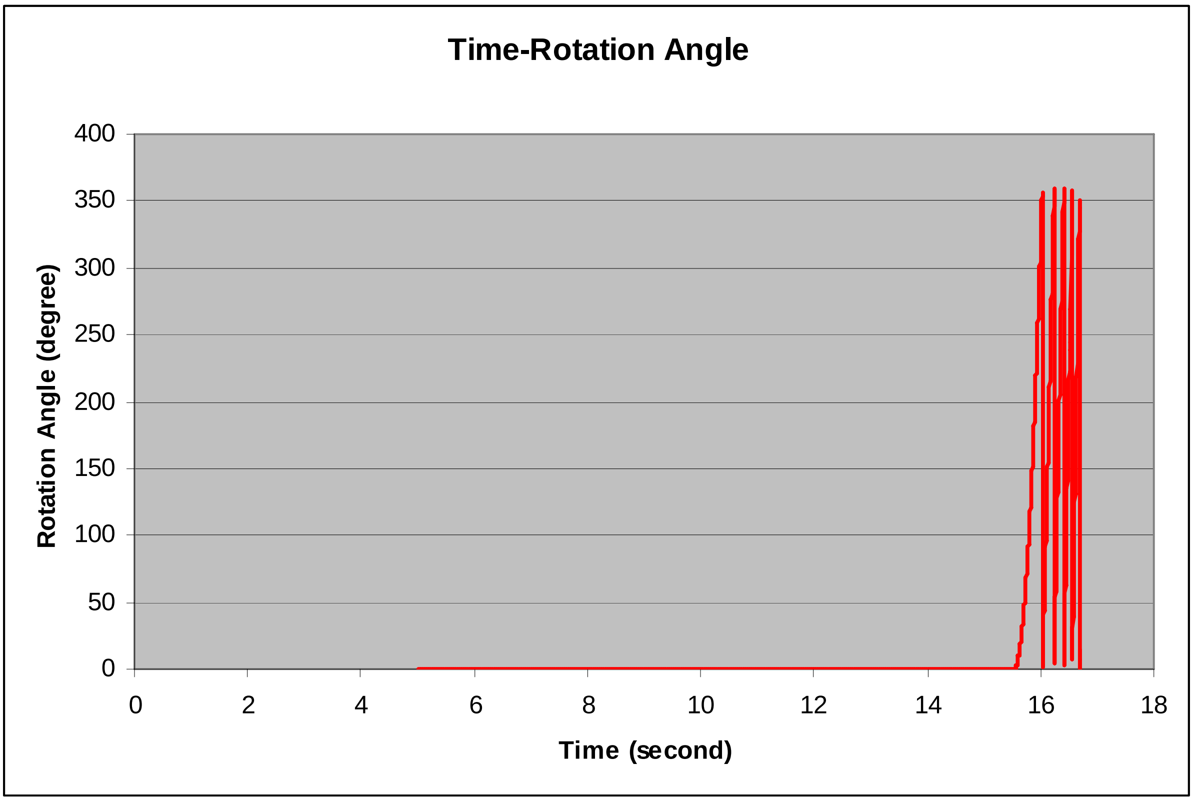
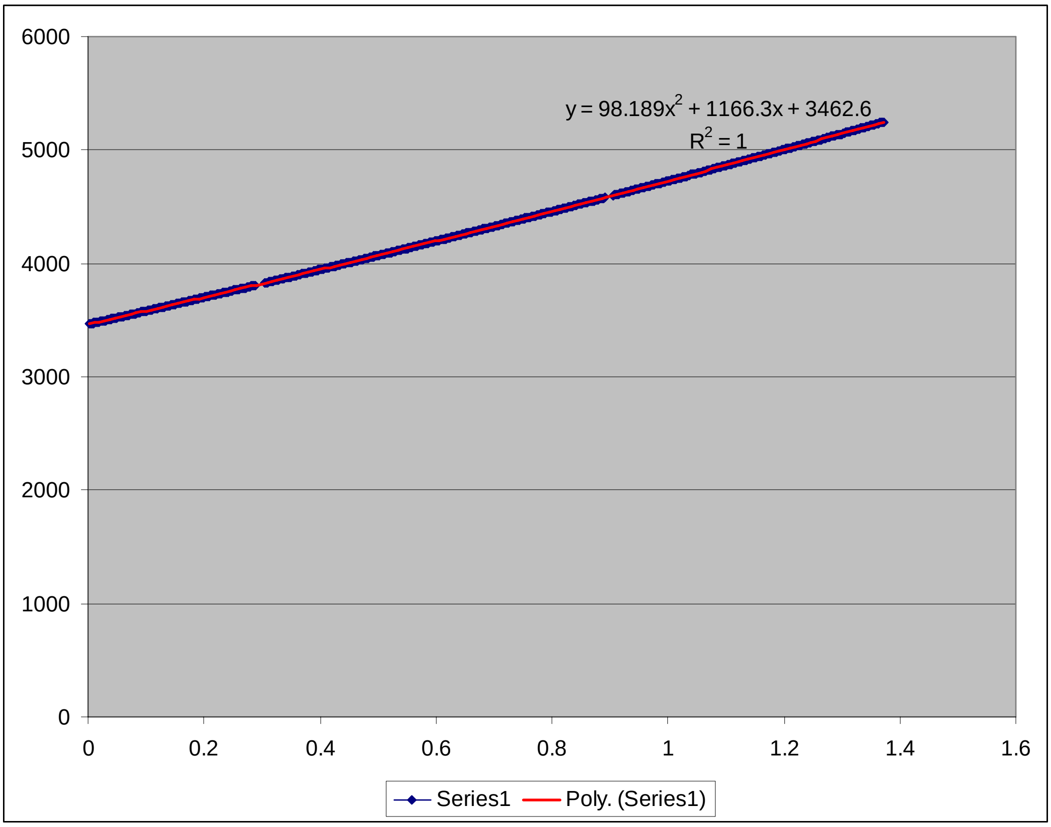


Figure 8 Rotation angle in the torque balance experiment

In Unreal engine, we have both 'mass' and 'kMass' parameters. In UDN, 'kMass' is interpreted as mass density. So the volume, which is decided by both shape and draw scale, might also affect an object's mass. To find out how 'mass' (the unreal parameter), kMass, drawscale, and staticmesh (shape) affect an object's mass, a serials of force testing experiments were conducted for objects with different parameters. The results are summarized in the following table (see Figure 6,9~13). I get the same conclusion as Marco's that in terms of Karma behavior, an object's mass is totally decided by kMass!

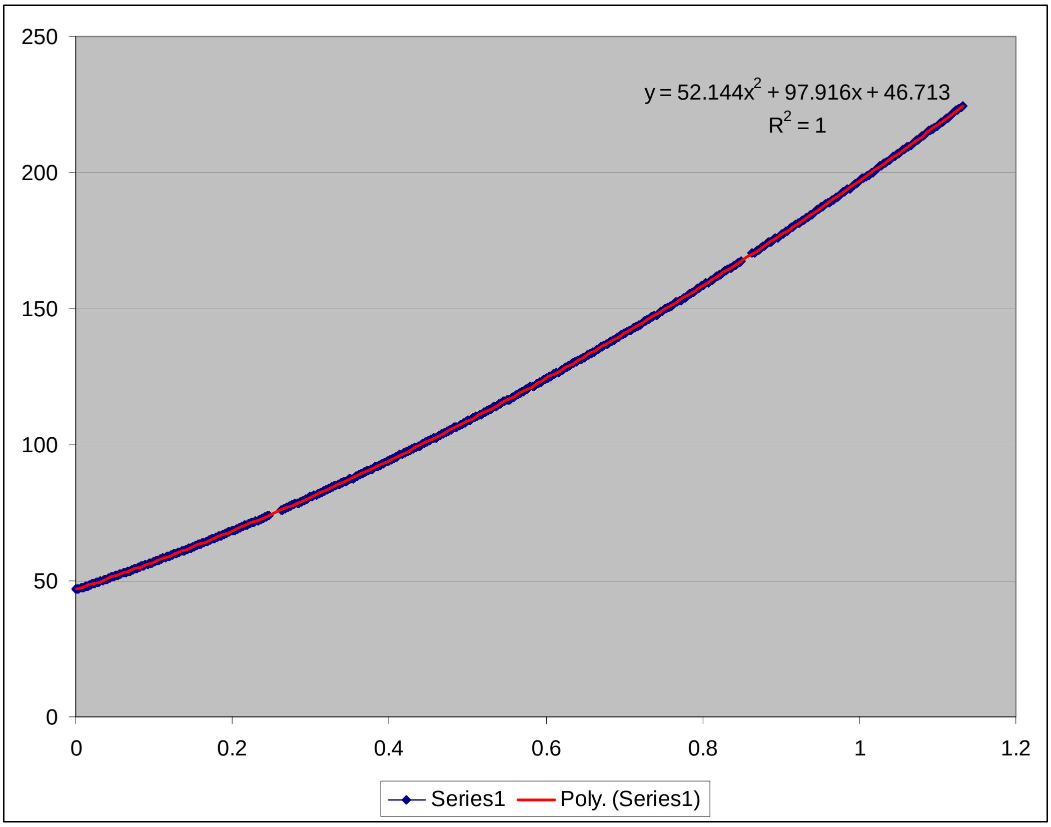
Acceleration (UU/s^2)	'mass'	kMass	drawscale	staticmesh
103	1	100	0.6	brick
196	1	50	0.6	brick
104	2	100	0.6	brick
197	2	50	0.6	brick
198	1	50	0.3	brick
206	1	50	1	sensor



$g_{setting}$ = 0 UU/s^2
mass = 1.0
kMass = 50.0

DrawScale = 0.6 StaticMesh=brick
F = 200 UU
a = 98.189*2 = 196 UU/s^2

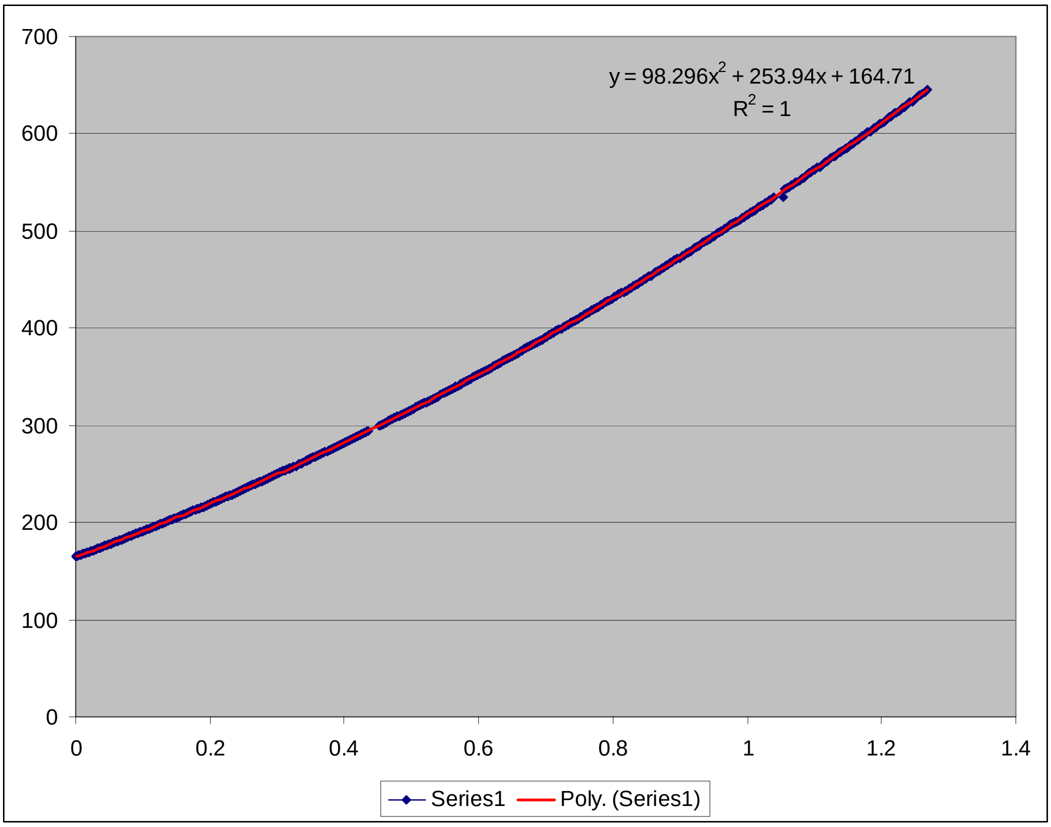
Figure 9 Force testing 3 with kMass = 50, mass=1



$g_{setting}$ = 0 UU/s^2
mass = 2.0
kMass = 100.0

DrawScale = 0.6 StaticMesh=brick
F = 200 UU
a = 52.144*2 = 104 UU/s^2

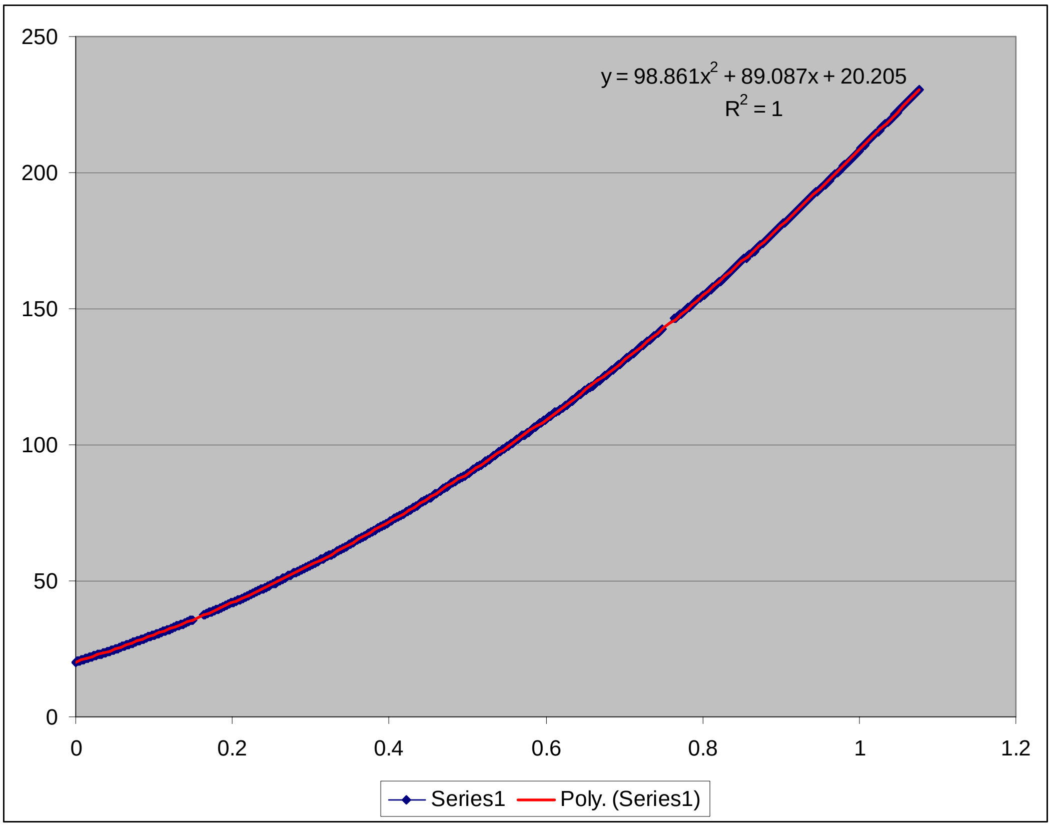
Figure 10 Force testing 4 with mass=2, kMass=100



$g_{setting}$ = 0 UU/s^2
mass = 2.0
kMass = 50.0

DrawScale = 0.6 StaticMesh=brick
F = 200 UU
a = 98.296*2 = 197 UU/s^2

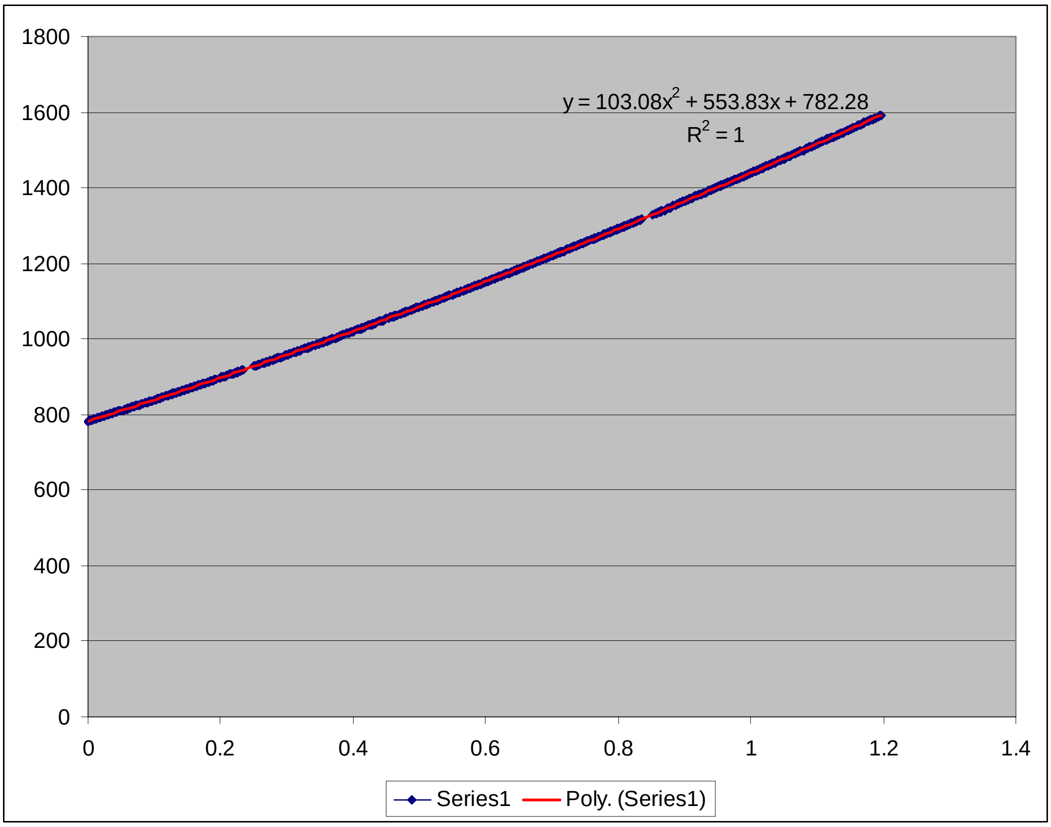
Figure 11 Force testing 5 with mass=2, kMass=50



$g_{setting}$ = 0 UU/s^2
mass = 1.0
kMass = 50.0

DrawScale = 0.3 StaticMesh=brick
F = 200 UU
a = 98.861*2 = 198 UU/s^2

Figure 12 Force testing 6 with drawScale=0.3



$g_{setting}$ = 0 UU/s^2 DrawScale = 1.0 StaticMesh=sensor
mass = 1.0 F = 200 UU
kMass = 50.0 a = 103.08*2 = 206 UU/s^2

Figure 13 Force testing 7 with another staticmesh (sensor)

Conclusion

In Unreal Engine, the unreal world and the Karma physical world use different time system. Counting this time scale, we get consistent length scale, 50uu=1m, from both our observation and the UDN document. On the other hand, unreal uses a simple 1:1 scale in the dynamic simulation. That is 1 uu mass = 1 Kg, 1 uu force = 1N, and 1 uu torque = 1 Nm. There is no fixed gravity acceleration in unreal. This value is controlled by PhysicsVolume. By default, it's set as Gravity=(0,0,-950). The scale used in Unreal is summarized below:

The world scale:
1.1GameSpeed uu time = 1 second
50 uu = 1 meter
1 uu mass = 1 kg
1 uu force = 1 N
g = 19 m/s^2 or anything we set in the game

As an example of applying these scales in unreal world, I did a 'complex' experiment. An object with 1 UU mass was put in the default PyhsicsVolume with extra against gravity force F=9UU. Using Newton's law, we can predict the falling acceleration will be:

$$\begin{aligned} a_{real} &= (mg-F)/m \\ &= (1\ UU_{mass} * 950\ UU_{length}/s^2 - 9\ UU_{force}) / 1\ UU_{mass} \\ &= (1\ Kg * 950/50\ m/s^2 - 9\ N) / 1\ Kg \\ &= (19\ Kg\ m/s^2 - 9\ Kg\ m/s^2) / 1\ Kg \\ &= 10\ m/s^2 \end{aligned}$$

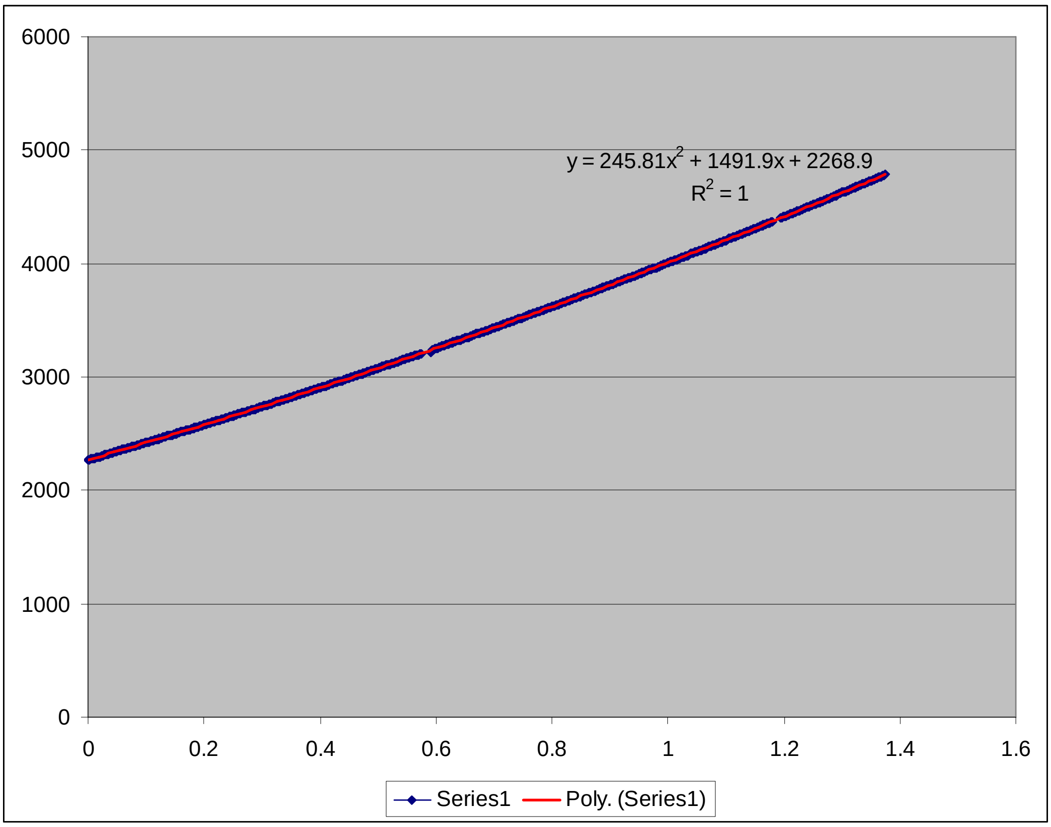
If we observe the falling activity in computer system time, the acceleration should be

$$a_{computer} = a_{real} = 10\ m/s^2 = 500\ UU/s^2$$

If we observe the falling activity in game time, then with the time scale, we will get

$$a_{game} = a_{real} / TimeScale^2 = 10\ m/s^2 / 1.1^2 = 500\ UU/s^2 / 1.21 = 413\ UU/s^2$$

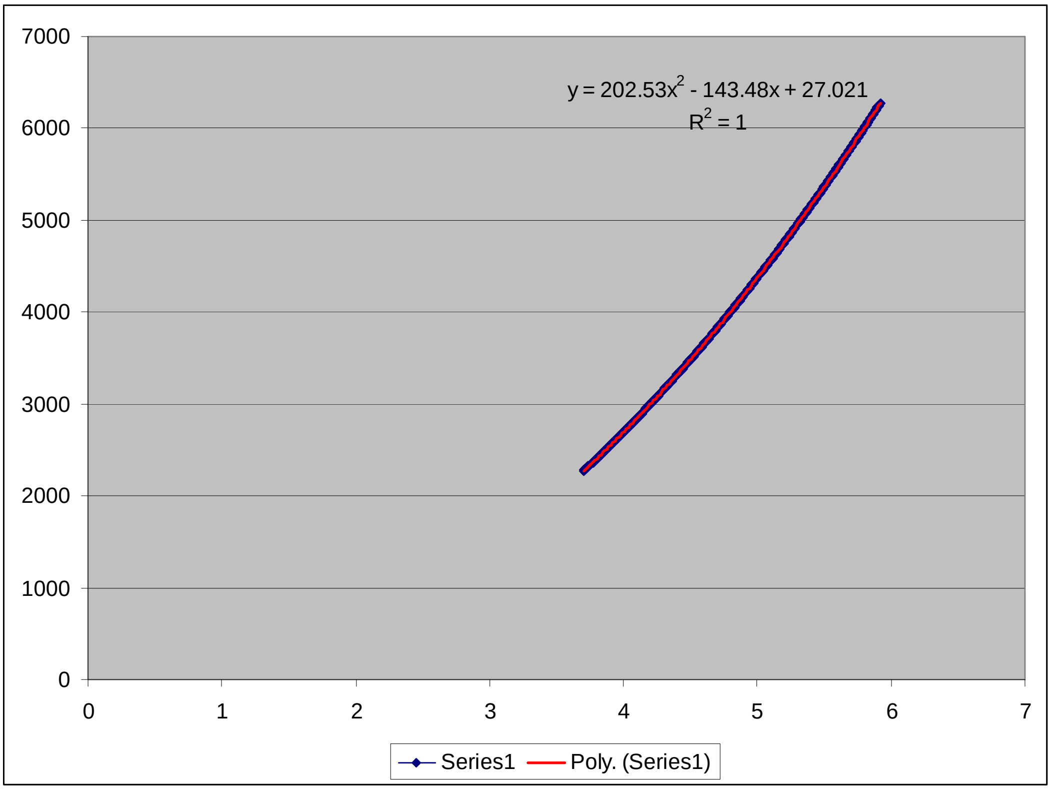
Figure 14,15 show the measured results. In computer system time, we get a = 492 UU/s^2 that is close to the predicted 500 UU/s^2. In game time, a = 405 UU/s^2. It's also close to the calculated value 413 UU/s^2.



$g_{setting}$ = -950 UU/s^2
mass = 1.0 UU
kMass = 1.0 UU

DrawScale = 0.6 StaticMesh = brick
F = 9 UU
$a_{computer}$ = 245.81*2 = 492 UU/s^2

Figure 14 The falling testing in computer time with F = -9 N



$g_{setting}$ = -950 UU/s^2
mass = 1.0 UU
kMass = 1.0 UU

DrawScale = 0.6 StaticMesh = brick
F = 9 UU
a_{game} = 202.53*2 = 405 UU/s^2

Figure 15 The falling testing in game time with F = -9 N