Realistic Sniper and Ballistics

Inan Evin, 2021

https://inanevin.com

Please prefer to use the online documentation at

https://rsb.inanevin.com

Welcome

Thank you for purchasing Realistic Sniper and Ballistics System (**RSB**). Please do rate and review the package on Unity Asset Store! If you have any questions, recommendations or any future requests, please reach me through the Support Discord or my e-mail address.

What is it?

Realistic Sniper and Ballistics System (**RSB**) is a toolset that includes many features/functionality you would need while developing a first-person / sniper game. Rather than being a template for a single genre, RSB provides back-support for many possible outcomes. Almost all the systems in RSB are decoupled from each other, and a robust event system makes it possible to integrate RSB into your own projects and only use those parts that are really needed! Thus, RSB can be used for third-person, first-person, or basically any genre of game that includes realistic projectile simulation.

About This Document

This is the official documentation where you can find examples, tutorials, features and the API of **RSB.** Please browse to the categories and pages from the menu on the left.

Introduction

Feature List

- **Centralized System:** By including a single singleton instance to your scene, you get access to all RSB features. You can write a single line of code to fire any type of bullet from anywhere in your scene. Use the included prefabs to create a first-person system out of the box, or include RSB in your scene and use it with your own systems!
- **Event Based System:** RSB works completely on events, so it becomes extremely easy to listen to bullet hit events, or bullet time events to trigger custom game code.
- **No Rigidbodies:** Bullets fired with RSB works completely with raycasts, meanwhile providing you with an extremely accurate real-life simulation precision. No objects being spawned, no rigidbodies are being fired. Thus, firing a bullet, even though it's simulated by many physics factors, is extremely cheap.
- **Realistic Bullet Ballistics:** Bullets are affected by: gravity(bullet drop), air resistance (multiple drag models w/ adjustable coefficient), spin drift (barrel twist), wind speed, air pressure and of course time.
- **Bullet Properties & Presets:** By using Scriptable Objects, you can create a custom bullet of your own just with couple of clicks. You can adjust the bullet's weight, diameter, drag coefficient and other physical properties if you want to simulate a specific type of a bullet. On top of that, RSB provides you with an extensive set of presets derived from real cartridges, such as 9mm, 7.62, 556, .338 rem, .40 ACP etc.
- **Custom UI:** An extensive set of custom UI tools to set up your own bullet properties, design-time trajectory simulator to visualize the bullet's trajectory in a graph on Unity editor itself.
- **Surface Ballistics:** Bullet penetration and ricochet, based on bullet's kinetic energy & velocity. Completely customizable interface to simulate various surfaces such as; wood, steel, metal, plastic etc. along with the presets of said surfaces.
- **Bullet Tracers:** Visualize your bullets, enable bullet tracers and define your tracer prefab to see your bullet's trajectory.
- **In-game Trajectory Debugging:** Enable trajectory debugging to visualize the whole bullet trajectory with line renderers.
- **Dynamic Scope System:** A dynamic canvas based scope system showing bullet drops depending on the selected bullet as well as how the bullet will be affected by the wind.
- **Bullet Time Cameras:** A powerful custom UI based tool to create your own bullet time camera effects without writing a single line of code. 10+ bullet time camera presets are included. Additionally, RSB provides a robust API to write your own bullet time camera effects if desired.

- **Player Controller:** RSB includes an example first-person player controller ready to be used in your own projects along with a smooth first person camera.
- **Motion Controller:** On top of the player controller, RSB also includes a motion controller script to create head-bob effects, camera/gun swaying, effects to simulate breath sway and a weapon positioner system to handle the position of the weapon depending on the player state (running, walking, aiming etc.)
- **Weapon Controller:** RSB includes an example weapon controller with an ammunition counter ready to be used for any type of weapon, especially for a bolt-action sniper rifle.
- **Mobile Controls:** RSB is mobile ready, and includes example scenes demonstrating how mobile controls can be used with its FPS controller.
- **Full Demo Scenes:** Full demo scenes demonstrating all features of RSB are included.
- **And much more utilities!**

Why/How is it used?

You can use the full features of RSB, or a combination of them, to create games such as:

- A first-person sniper game that includes realistic bullet trajectories, penetration, ricochet and bullet time, completely out of box and mobile ready.
- Any type of shooter game, first-person or third-person, that includes realistic bullet ballistics. You can use RSB's ballistic simulation feature to fire completely accurately simulated bullets & listen to their events to know what was hit and how. Thus, you can use RSB in any type of shooter game.
- Any type of game that requires realistic projectile simulation, such as tank battle games.
- Well, basically anything, every system in RSB is decoupled so you select what to use.

Quick Setup

Import RSB into your project. Then navigate to **InanEvin>Realistic Sniper and Ballistics>UNPACK** folder. You will see 3 packages for each of the render pipelines (**standard, urp/lwrp and hdrp**). Unpack the one your project uses, you can delete the others. Then, navigate to the folder **DemoScenes** and play any demo scene you want. You will see:

- **Full Demo:** Demonstrating all RSB features in a first-person sniper game fashion.
- **Mobile Demo:** Demonstrating RSB features in a first-person sniper game fashion with a mobile ready flat scene.
- **EmptyPlayerControllerDemo:** Demonstrating the first-person controller that is included in RSB, with complete movement and motion such as head-bob, camera sway etc.
- **MinimalDemo:** Demonstrating the usage of RSB without a player controller. Perfect if you want to integrate RSB into your own controllers and not use the provided ones.

FullDemo demonstrating e.g bullet ricochet & penetration

Mobile demo with bullet time, recorded in Galaxy A70

Getting Started

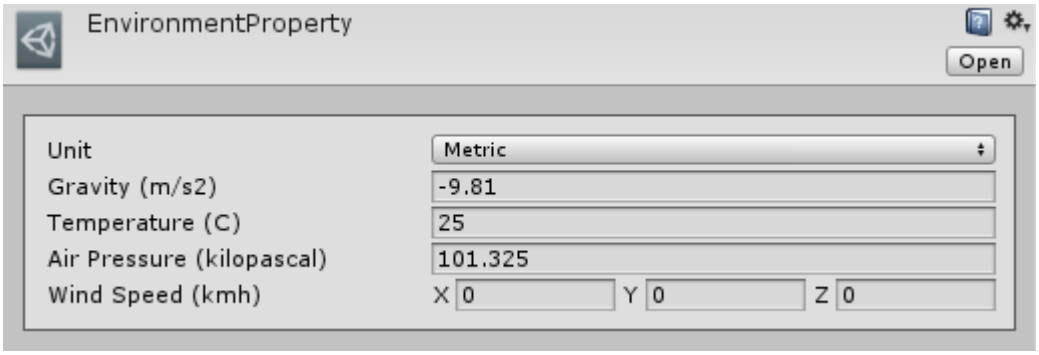
Getting Started

This section assumes you'd like to learn about how RSB works step by step, this way you can use RSB features with a complete control over them, e.g, to integrate RSB into your own systems & players. So let's create an empty scene and work on that to learn how RSB works. The empty scene should contain a directional light (if desired) and a single Main Camera setup at 0,0,0 as position.

RSB works by using two scriptable object assets as means of determining the simulation settings. These are **EnvironmentProperties** and **BulletProperties** assets. So first, let's setup these files and make them ready to be used within our system.

Environment Properties

An Environment Properties asset determines your scene's global environment settings which will be used during simulation of a bullet. To create one, right click anywhere in your project file, go to **RSB>Environment Properties**. This will create a new environment properties scriptable object asset file in your project folder.



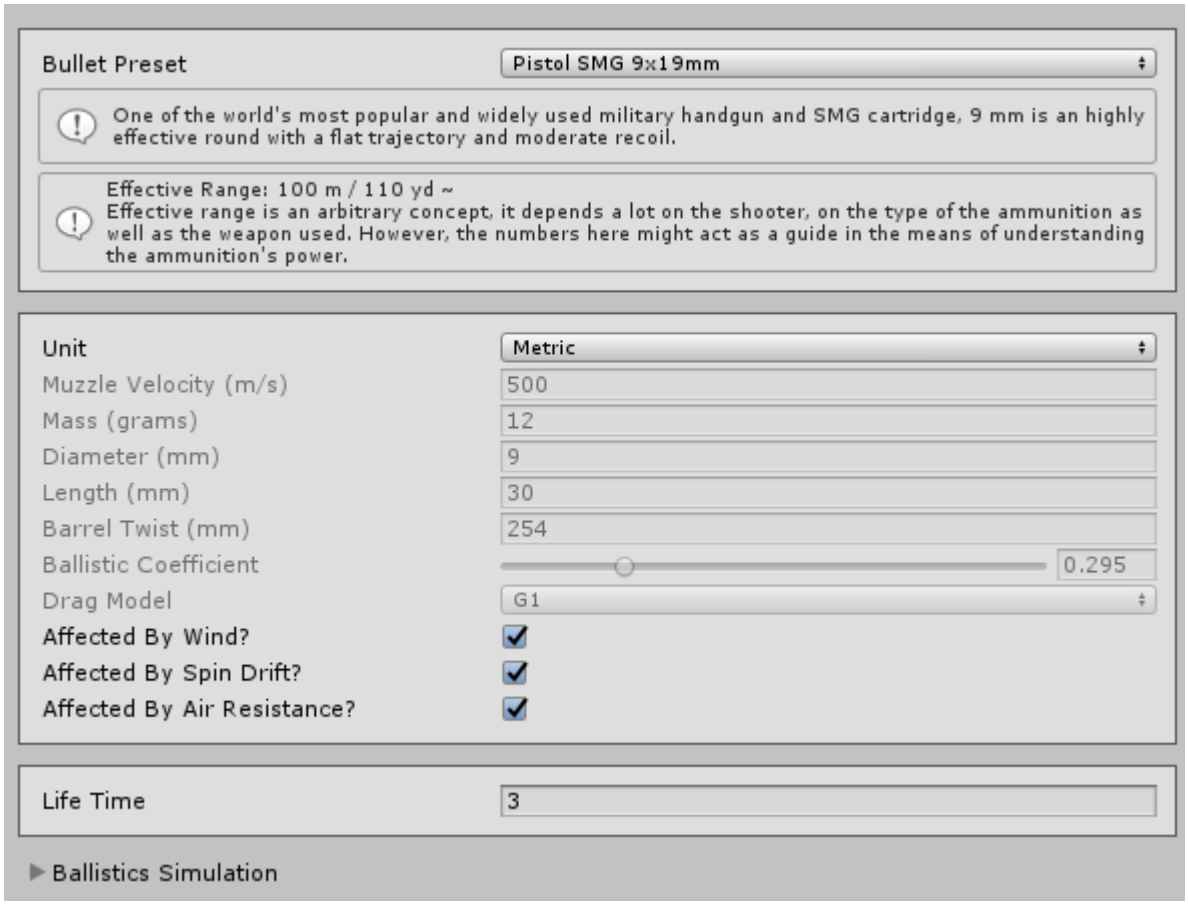
Environment Properties scriptable object asset file.

As you will see, environment properties determine your scene's global settings such as gravity, temperature, wind speed etc. Your bullet simulation will be affected by these settings. Most of the time, you can leave gravity, temperature and air pressure in it's default settings but play with the wind speed to achieve the desired wind effect. After we get our environment property ready, we need to create a **Bullet Properties** asset so we can fire that bullet.

> Wind Speed property is accessible in runtime, so we can for instance create a dynamically changing wind condition during our game.

Bullet Properties

A Bullet Properties asset determines the simulation settings of your virtual bullet. To create one, right click anywhere in your project file, go to **RSB>BulletProperties**. This will create a new bullet properties scriptable object asset file in your project folder.



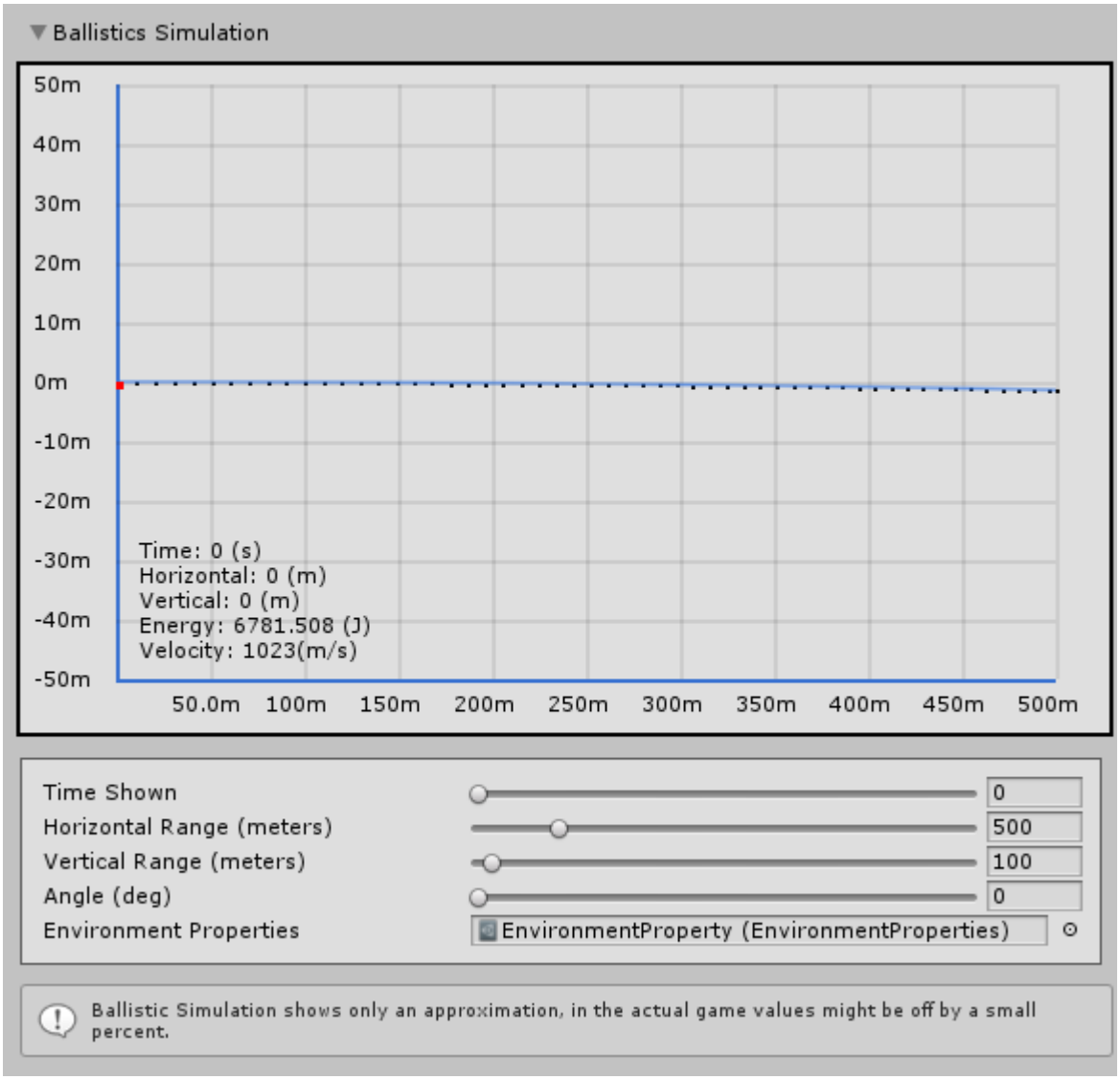
Bullet Properties scriptable object asset.

First thing you will see is the **Bullet Preset** dropdown. Here, you can select from available bullet presets, such as rifle ammunition or sniper rifle ammunition, derived from real cartridges. You can also see information about the preset in the info boxes below. The properties are pretty self-explanatory, you can customize your bullet by selecting the **Custom** preset & setup your muzzle velocity, mass, diameter etc. other physical properties of your bullet. Most important values in here are **Muzzle Velocity**, and the **Ballistic Coefficient**. Muzzle velocity determines the speed/power of your bullet. Sniper rifle cartridges usually have muzzle velocities between **700 m/s** and **1500 m/s**. Ballistic coefficient determines your bullet's power to punch through air resistance. Higher the

coefficient, bullet is less affected by the air drag. Usually values between **0.2-0.8** are used. You can also determine whether your bullet will be affected by wind, spin drift or air resistance. Let's select the **SR338 Magnum** from the preset dropdown to use for this tutorial.

> You can create multiple BulletProperties asset files, each defining one type of a bullet you want to use in your game. You can either use presets, or set it to **custom** for setting your bullets physical properties yourself. However, it is a good idea to stick to realistic measurements and weights.

When you unfold the **Ballistics Simulation** foldout, you can visualize your bullet's trajectory in the editor. You will see below is that the simulation requires an Environment Properties asset. Select the environment properties we just created in the section above, and you will see the trajectory of your bullet.



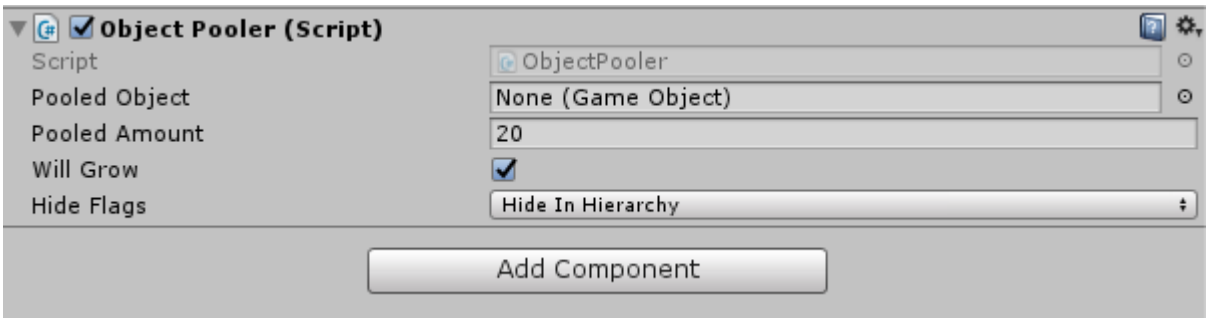
Ballistic Simulation showing bullet trajectory in design-time on Inspector.

This simulation gives you an idea about how powerful/accurate your bullet is. It does not account for spin drift or wind, but only calculates the trajectory based on gravity, temperature, air density and air resistance. You can play with the graph settings to visualize how much your bullet will drop in different distances. Once we have our **Environment Properties** and **Bullet Properties** asset files ready, now we can start using our system.

> The Environment Properties asset needed in Bullet Properties inspector is only needed for simulation purposes in the editor. You do not have to assign the field if you don't want to see the simulation in editor, it does not affect the gameplay at all. In gameplay, the Environment Properties asset will be used that was assigned to the instance of SniperAndBallisticsSystem.cs file, which will be explained in detail in the upcoming sections.

Object Poolers

RSB uses object poolers to handle spawning/instantiating prefabs that will be used during simulation. These prefabs can be bullet tracers, in-game trajectory renderers, hit particles etc. In order to safely handle object instantiation in an optimized manner, RSB makes use of **Object Poolers.** Create a new empty game object in your scene and name it **Tracer Pooler.** We will later on use this object to pool **tracers**, visualized objects simulating bullet tracers. Add the component **ObjectPooler.cs** on it.

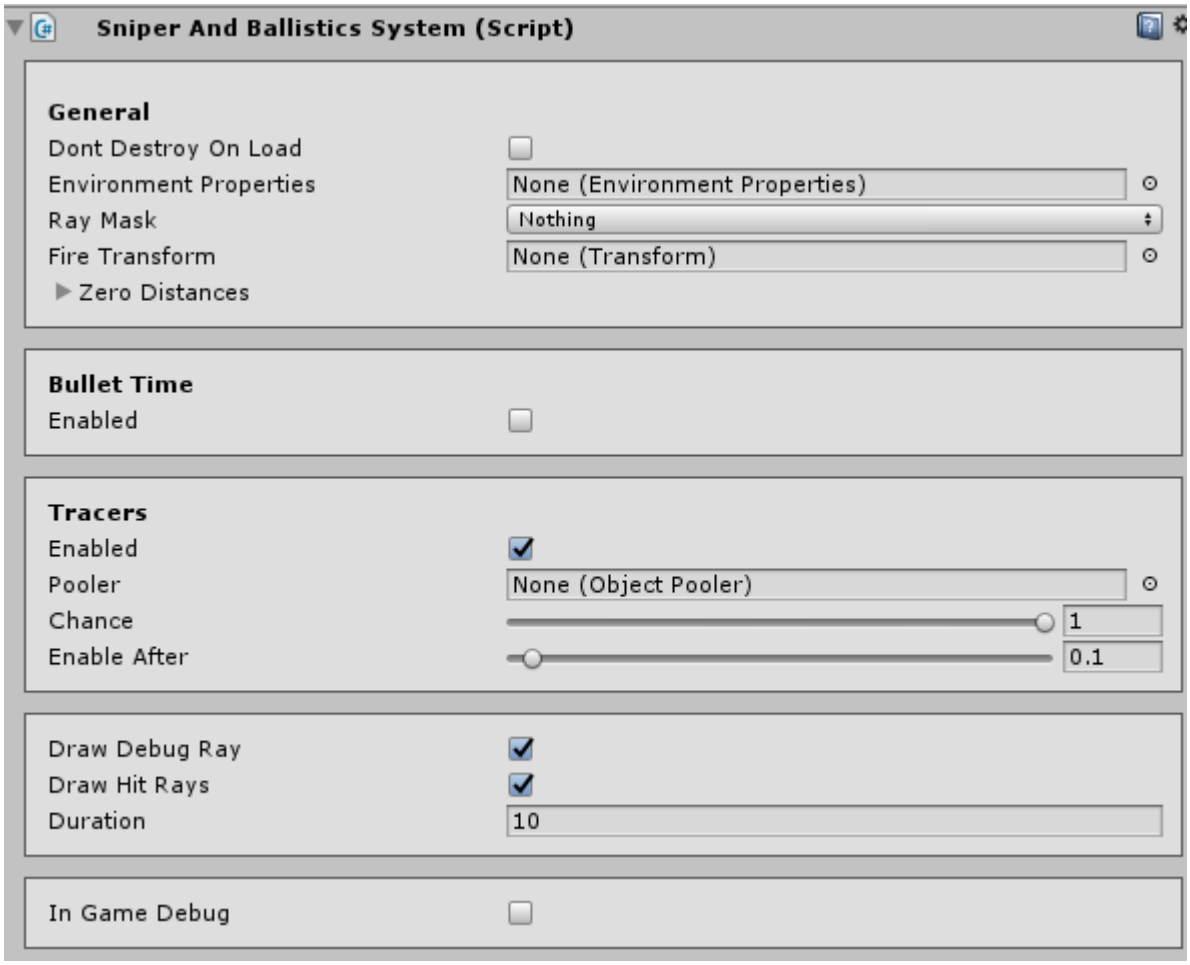


ObjectPooler.cs settings.

You need to assign the **Pooled Object** field as a prefab in your project folders. Navigate to **InanEvin>Realistic Sniper And Ballistics>Objects>BulletTracer** folder and drag & drop the tracer prefab into the slot. You can leave the other settings as default. What this will do is that when you start your scene, this script will instantiate **Pooled Amount** of bullet tracers in awake, disable them all and keep them in a list. Later on, our system will request a new tracer object whenever it is needed, and this script will return an available one from the list, without instantiating. This way, we delegate the instantiation of prefabs to the **Awake(),** saving us the runtime performance drop of instantiating. The same approach will be used on any other type of instantiation, for in-game trajectory renderers, hit effects, muzzle flashes etc.

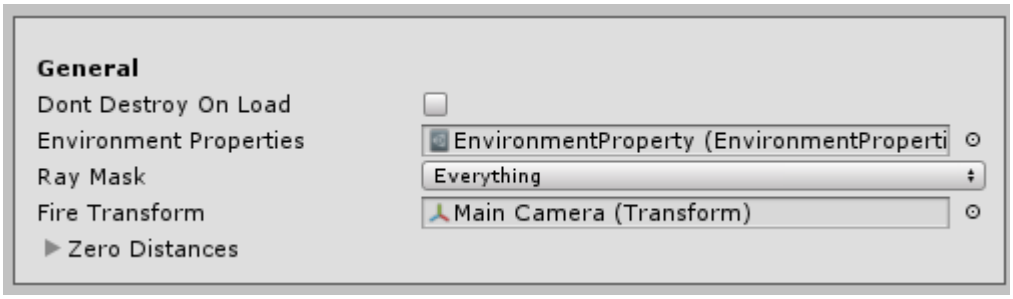
Sniper and Ballistics System

The central piece of RSB is a **SniperAndBallisticsSystem** singleton instance. Without it, you can not use the ballistics simulation. Create an empty game object, name it **Sniper and Ballistics** and attach the script **SniperAndBallisticsSystem.cs** on it. This is what you will see:



SniperAndBallisticsSystem.cs instance in inspector when attached to an object.

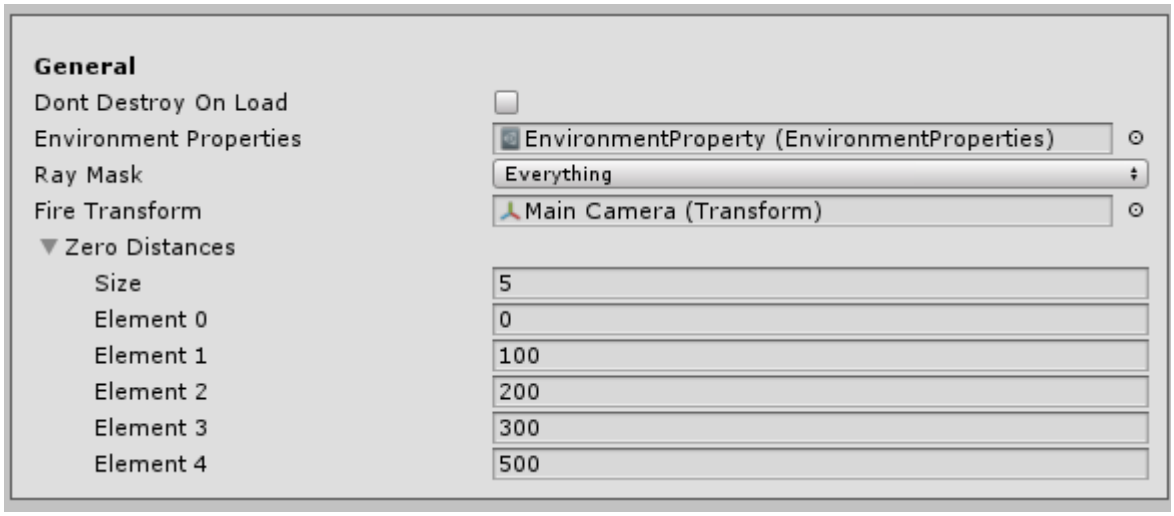
First thing you will need to do is to assign an environment property to the instance. Drag & drop our environment properties asset file to the Environment Properties field. Now we'll have to setup the **Ray Mask**, which defines which layers will the raycasting be applied to. Set it to **Everything** for now.



General Settings for SniperAndBallisticsSystem.cs

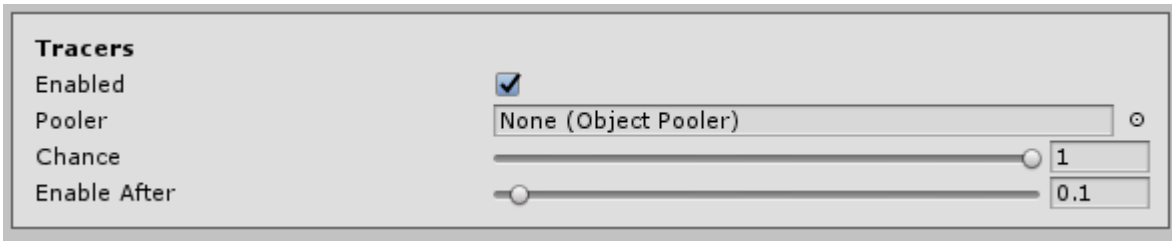
Fire Transform determines where your shots are coming from. It is natural to think that this should be an object at the end of the weapon's barrel etc. however in most cases this is your **main camera**. Because the **Fire Transform** will determine where your raycast is originating from, and in most cases you want to fire according to what your camera sees. Effects such as muzzle flashes, or your bullet time bullet's starting position is determined separately, and that can be e.g an empty game object at the end of your weapon's barrel. So let's assign the Main Camera object to it in our empty scene.

Next thing is setting up **Zero Distances**. If you do not know about **zeroing**, I would definitely suggest going for a quick google search about it. Here the **Zero Distances** list is basically the available list of zero distances your game/scene should have. You setup this list once, and all other systems will pre-calculate their necessary simulation values according to this list. Then, you'd be able to cycle the current selected zero distance to setup your bullet's zeroing, making it hit the target at the selected distance directly.



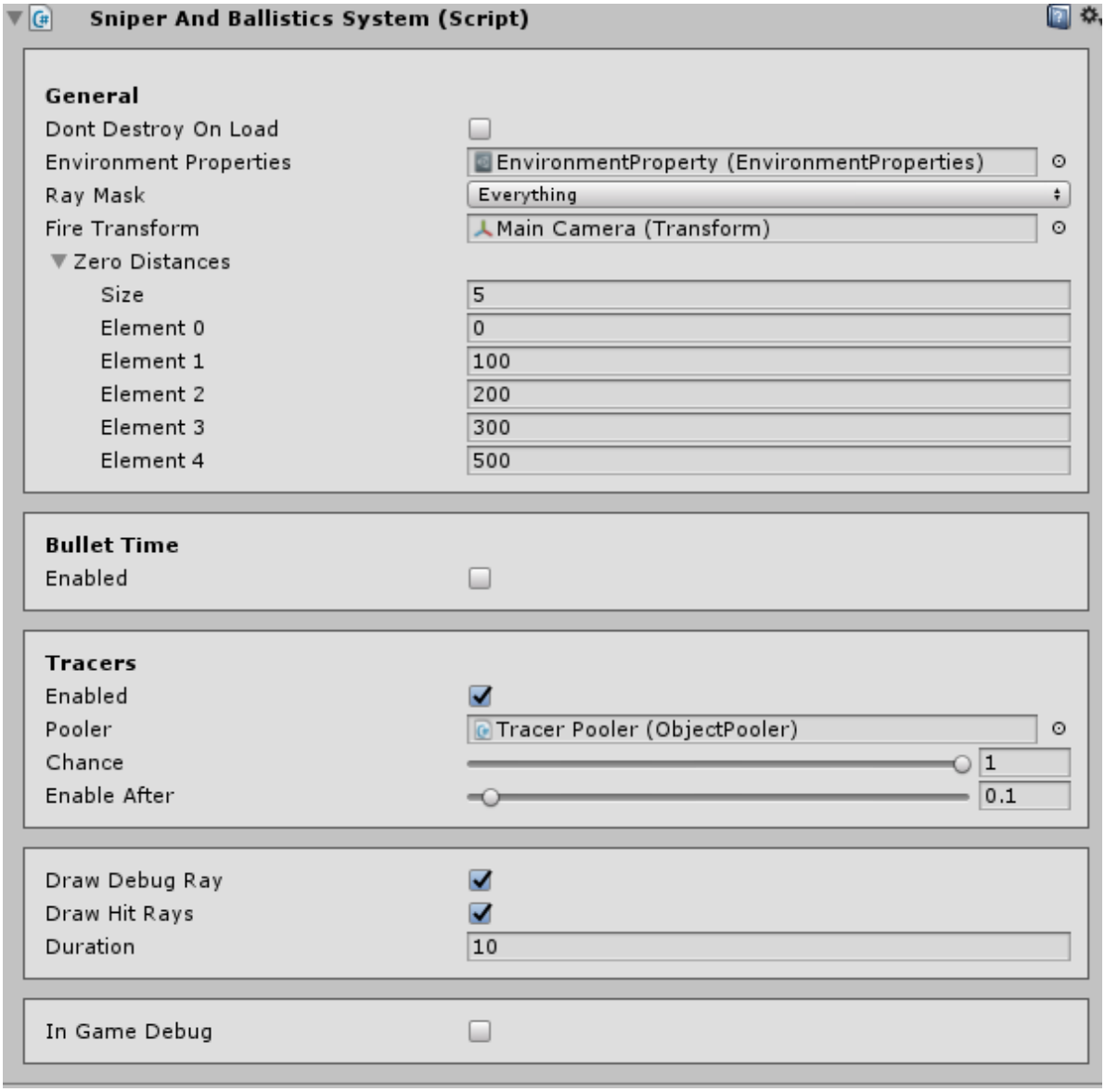
A proper Zero Distances list set in General Settings of SniperAndBallisticsSystem.cs

The system is hard-coded to have the first element of this list as 0. So this is how it would look like to setup a proper zero distance list. It is not recommended to go over 500 meters, in real-life most scopes can not adjust for such distances, and the simulation becomes less accurate the higher you go above 500 meters. This is mostly depending on your bullet's muzzle velocity and air resistance as well. For instance, 1000 meters of zero distance on a 9mm bullet won't be as accurate as it will be on a 7.62mm cartridge.Now let's leave the **Bullet Time** disabled for now, and navigate to the **Tracers** section. When tracers are enabled, a tracer prefab will be spawned in each time you fire and it will follow the raycast path, simulating the effect of a bullet tracer.



Tracers Settings in SniperAndBallisticsSystem.cs

Chance parameter defines what is the random chance of getting a tracer spawn on each time you fire, and **Enable After** parameter defines exactly how many seconds after firing the tracer will be enabled. Let's leave them to default for now, but we need to assign a pooler instance that will be responsible for spawning the tracers. In the previous section **Object Poolers**, we have created an empty game object called **Tracer Pooler,** attached the pooler script on top of it & assigned the object to be pooled. Now let's drag and drop that object pooler instance to the **Pooler** field of **tracer settings.**



SniperAndBallisticsSystem.cs instance ready to be used within the scene.

This is how our instance will look at this point. You can leave **DrawDebugRays** and **DrawHitRays** options enabled so that we can see the bullet's path in Scene view. If you want an in-game debug, you can also enable it, setup an in-game debug pooler and use the **TrajectoryRenderer** prefab under **Objects** folder. Same logic as the tracers, but we will skip that for now. Since we've setup our SniperAndBallisticsSystem at the most basic level, we are now ready to fire a ballistics bullet!

> Under **InanEvin>Realistic Sniper and Ballistics>System Prefabs** folder you can find a prefab called **Sniper And Ballistics System**. This prefab is a ready-to-use SniperAndBallisticsSystem.cs instance with a bunch of poolers, and hit listeners

> attached as a child. For now, we **won't** use it for this tutorial, but definitely keep it in mind that you can use it in your own scenes.

Firing a Bullet

In order to fire, we need to call a single method on **SniperAndBallisticsSystem** instance. First let's create an empty game object and name it **Tutorial.** Then create a new script, name it **Tutorial.cs** and then attach this script to the Tutorial object in your scene. Open up the **Tutorial.cs** script. In order to use RSB classes, we first need to include a using **IE.RSB** directive at top of our script.

RSB-Tutorial1.cs

https://gist.github.com/36d755a8e7763655a08388c75ee5640f.git

Then, inside our class, we will need to reference the bullet properties asset file we've created to use it while firing.

RSB-Tutorial2.cs

https://gist.github.com/bdb394e22933ef181bf30d4048bedd6f.git

In order to be able to fire a bullet, we first need to **activate** it. Activating a bullet triggers a set of functions to pre-calculate some bullet variables such as it's drops or zero angles, which are needed during

simulation. By activating the bullet first, we avoid calculating these variables in runtime, saving us a performance. To do so, we will need to call a method in SniperAndBallisticsSystem instance. This is a singleton instance, meaning that it can be accessed from anywhere in your scene as long as one instance is attached to an object in the scene, which is the case for us.

RSB-Tutorial3.cs

https://gist.github.com/ca3e8dcc2f48c26861a7c82cc01c33bd.git

It is a good idea to activate the bullets in **Start** function, since we make sure SniperAndBallisticsSystem singleton instance will be definitely available to use in Start. We call the **ActivateBullet** method, pass in our bullet properties instance. You need to do this in

start function for every type of bullet property you plan to use in your scene. Usually this will be only for a single bullet, but for instance if your game has different ammunition on different weapons, you can write a script that will create a list of bullet properties references and iterate over them to call the function above in Start for all of them.

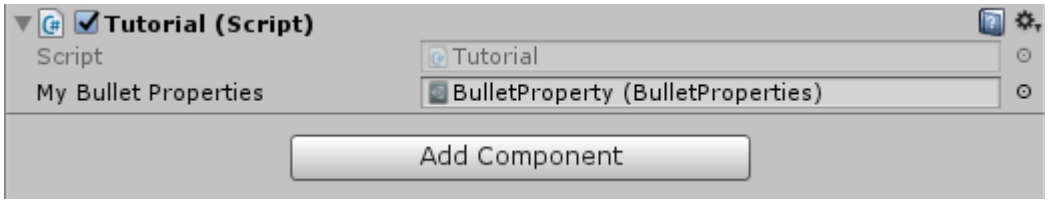
Next, let's fire this bullet! Let's setup a block in our update so that we can fire when we press **Space** key.

RSB-Tutorial4.cs https://gist.github.com/35160235b923131fc8dc5cbd34ea5efd.git

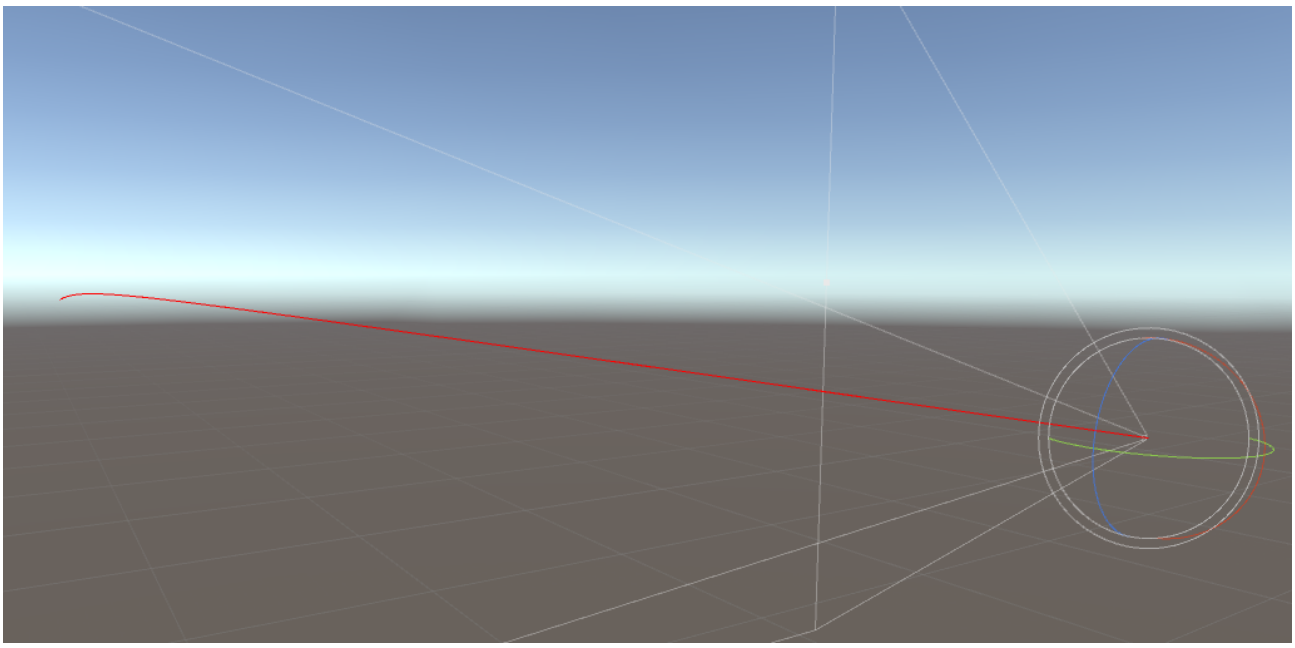
Now, only thing we need to do to call FireBallisticsBullet method to fire our bullet defined by myBulletProperties.

RSB-Tutorial5.cs https://gist.github.com/16974fb64069fa7283b05dfb443f5916.git

And that's it! Now if we go to the **Tutorial** object we have attached this script, and drag & drop our bullet properties asset file as the reference to **myBulletProperties** variable, we will be good to go!



If we play the game and hit **space** key the system will fire from the **main camera,** because that was what we have attached as the **Fire Transform** variable in SniperAndBallisticsSystem instance. We can see the bullet trajectory in scene view since we have enabled ray debugging.

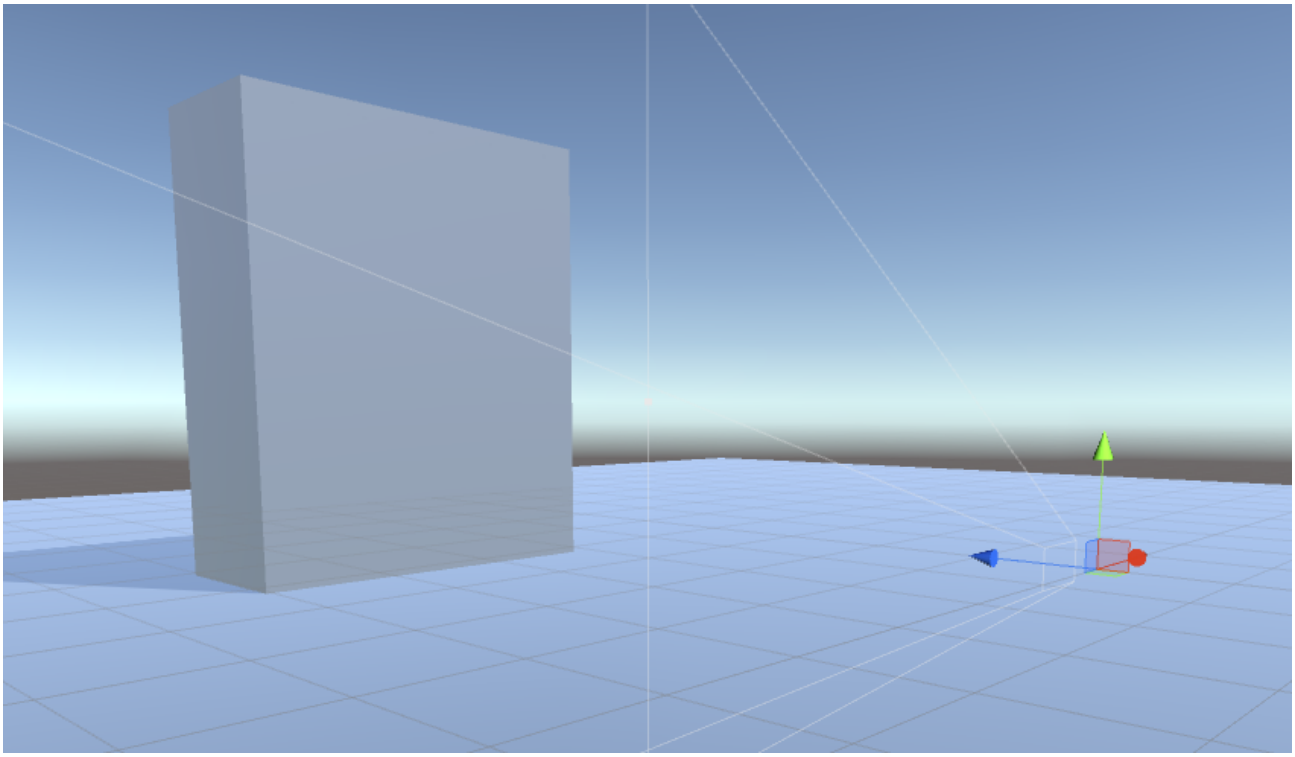


Scene view showing the trajectory of the bullet fired from main camera.

We have fired our first bullet which is ballistically simulated! Now let's learn how to listen to hit events for this bullet, so that we can cast particles, or apply forces to rigidbodies that are hit.

Listening to Hit Events

Let's first create a new plane object, place it in position **0,-1,0** and scale it to **5,1,5**. This will act as our floor. Then create a new cube object, place it to **0,2,10** and scale it to **5,6,2.** Remember our main camera was positioned at 0,0,0. The scene should look like this:



Main camera, floor & cube objects.

Now if you hit **space** key, you will see that the bullet hits the cube, and a green ray is casted upwards from the hit point. This is because **Draw Hit Rays** checkmark should be enabled in SniperAndBallisticsSystem.cs instance.

> In order for raycasting to detect objects, your objects need to have proper colliders on them. By default, the cube we have created will have a box collider attached.

Next, open up our **Tutorial.cs** script. RSB provides a robust event system to understand what's going on during the simulation. There are 5 events we can listen to in order to get information about what was hit and how:

- **EPenetrationInHit:** Called when the bullet hit's a surface that it has started penetrating.
- **EPenetrationOutHit:** Called when a bullet leaves a surface that it was penetrating.
- **ERicochetHit:** Called when the bullet ricochets off a surface.
- **ENormalHit:** Called when the bullet hits a surface, doesn't penetrate or ricochet.
- **EAnyHit:** Called for all hit types, this is what you will use in most cases.

In order to listen to these events, we first need to register to them. All the events send a single parameter of type **BulletPoint**, which is a class defining a point in bullet's travel path. It contains information such as the hit type, hit transform, normal, bullet's velocity, kinetic energy etc. In order to register to an event, we first need to have a method in our class that corresponds to the event's signature. In somewhere empty in our **Tutorial.cs** script, let's create a new method:

RSB-Tutorial6.cs https://gist.github.com/b0fb425e5bd83e3f28c3dc280cf59af8.git

Then we need to register this method as a callback to the **EAnyHit** event. It is a good idea to register to events in **OnEnable()** and deregister them in **OnDisable()** functions.

RSB-Tutorial7.cs https://gist.github.com/22afbefb69ae12a7f19188320afe699b.git

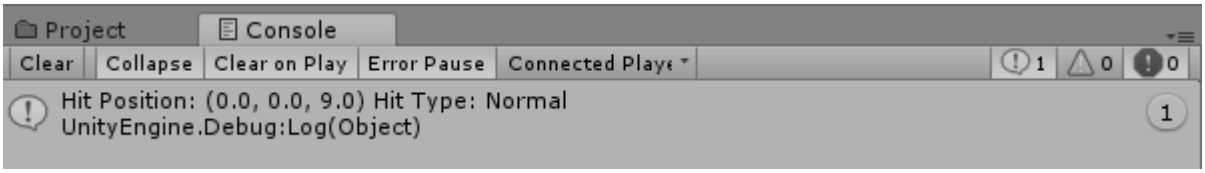
Now doing so, whenever something is hit by the simulation raycasts, our **OnAnyHit** method in **Tutorial.cs** will get called with information about the hit. Let's add a **Debug.Log** to print out some hit information.

RSB-Tutorial8.cs https://gist.github.com/a617114f862cdb8629e929ab53953e99.git

Her, we use the **m_endPoint** property inside **BulletPoint** reference to learn the hit position, then also print the hit type using **m_hitType** property. Now, our whole **Tutorial.cs** script should look like this:

RSB-Tutorial9.cs https://gist.github.com/a9ee3804d9158530ab08eedc907425b3.git

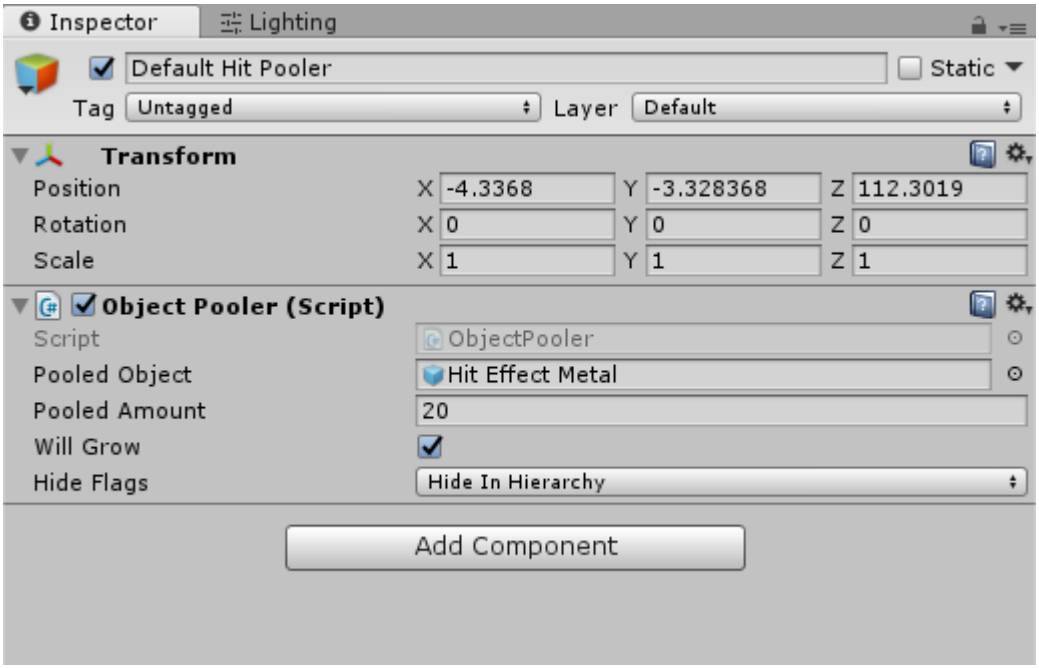
Now when you play the game, hit space key and watch the console, you should see the output:



By listening to hit events, you can for example write a hit listener that will apply force to the hit objects who has a rigidbody. An example of this can be found in **HitListenerApplyForce.cs** script under **InanEvin>Realistic Sniper and Ballistics>src>Hits** folder. Moreover, using the powerful structure provided by events, it is possible create particle managers who are responsible for listening to hit events, checking the hit object's tags or layers and spawning the appropriate particle effects. An example of this can be found in the same folder, in the **HitListenerSpawnParticle.cs** script. But for the sake of our tutorial, let's do this ourselves in the next page.

Spawning Particles on Hit

As mentioned before, RSB works with **Object Poolers** to manage object spawning. So it's a good idea to stick with the same methodology in our tutorials. Let's create a new empty game object, name it to **Default Hit Pooler**, then attach the **ObjectPooler.cs** script on this. By clicking to the prefab selection button next to the **Pooled Object** field, select the **Hit Effect Metal** prefab.



Default Hit Pooler object created, Object Pooler assigned, Hit Effect Metal is selected.

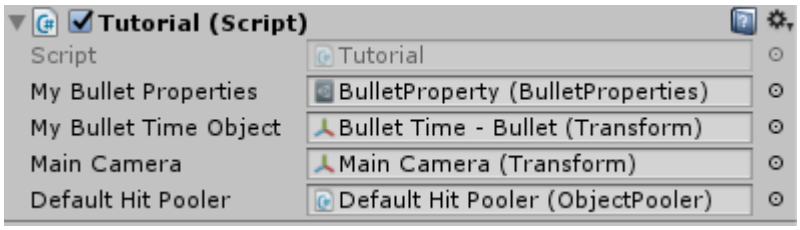
Now, let's add a field in our **Tutorial.cs** script to reference to this pooler.

RSB-Tutorial13.cs https://gist.github.com/48dad8629f2ce5da818695eb768645c1.git

Then, in our **OnAnyHit** method, we will get a new pooled object. Place it to the hit position, make it look at to the hit normal, and enable it.

RSB-Tutorial14.cs https://gist.github.com/519f6f4296e3782e0a395bee513279da.git

Do not forget to assign the **defaultHitPooler** variable on inspector of our Tutorial object.

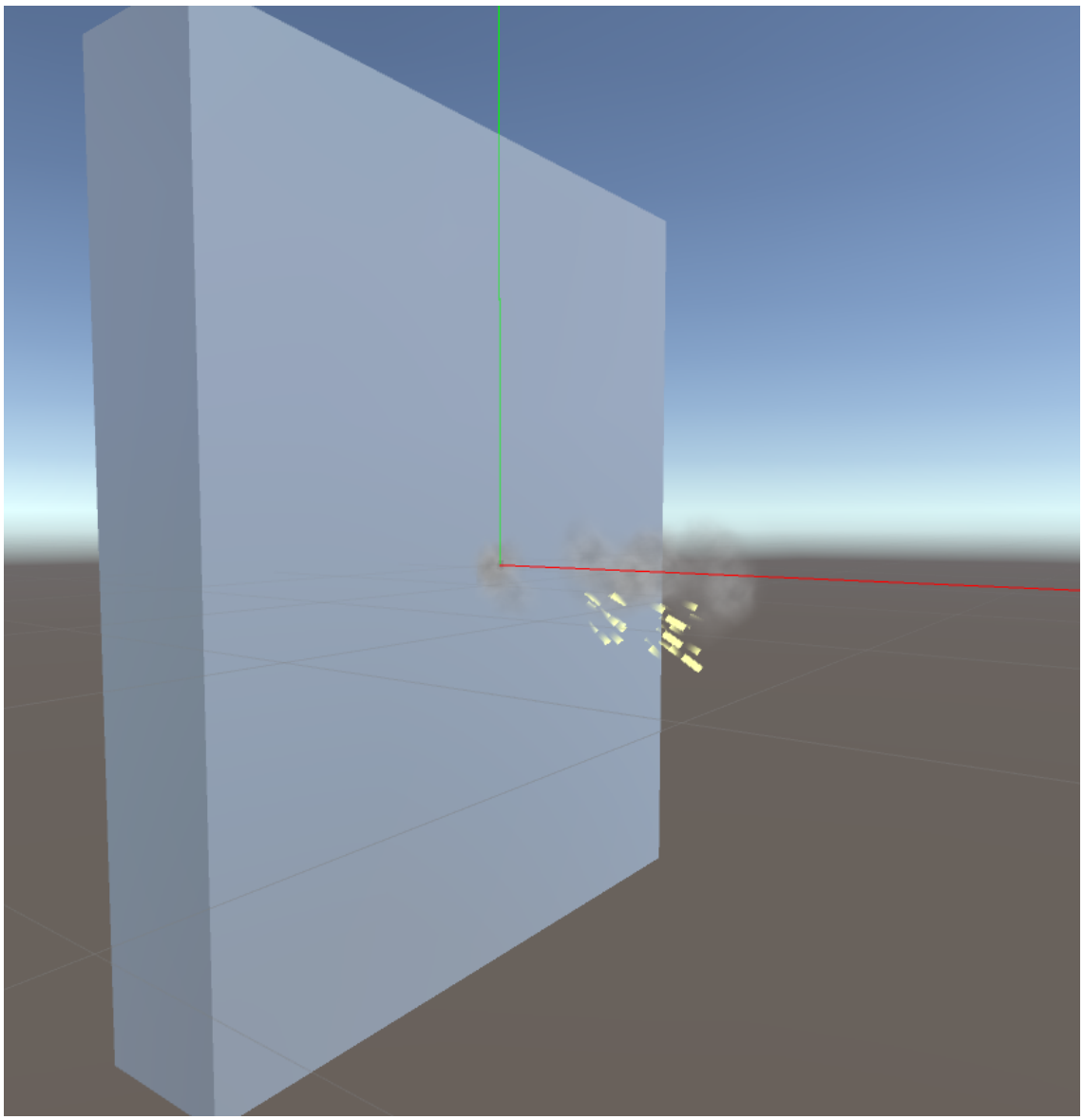


Default Hit Pooler assigned

So, now our whole **Tutorial.cs** script should look like this:

RSB-Tutorial15.cs https://gist.github.com/790c971941503896ae4cedfeaedf0159.git

Then, if we play and shoot, we will see the default dirt particle is spawned when the bullet hit's the target.



Metal hit particle gets spawned when the target is hit.

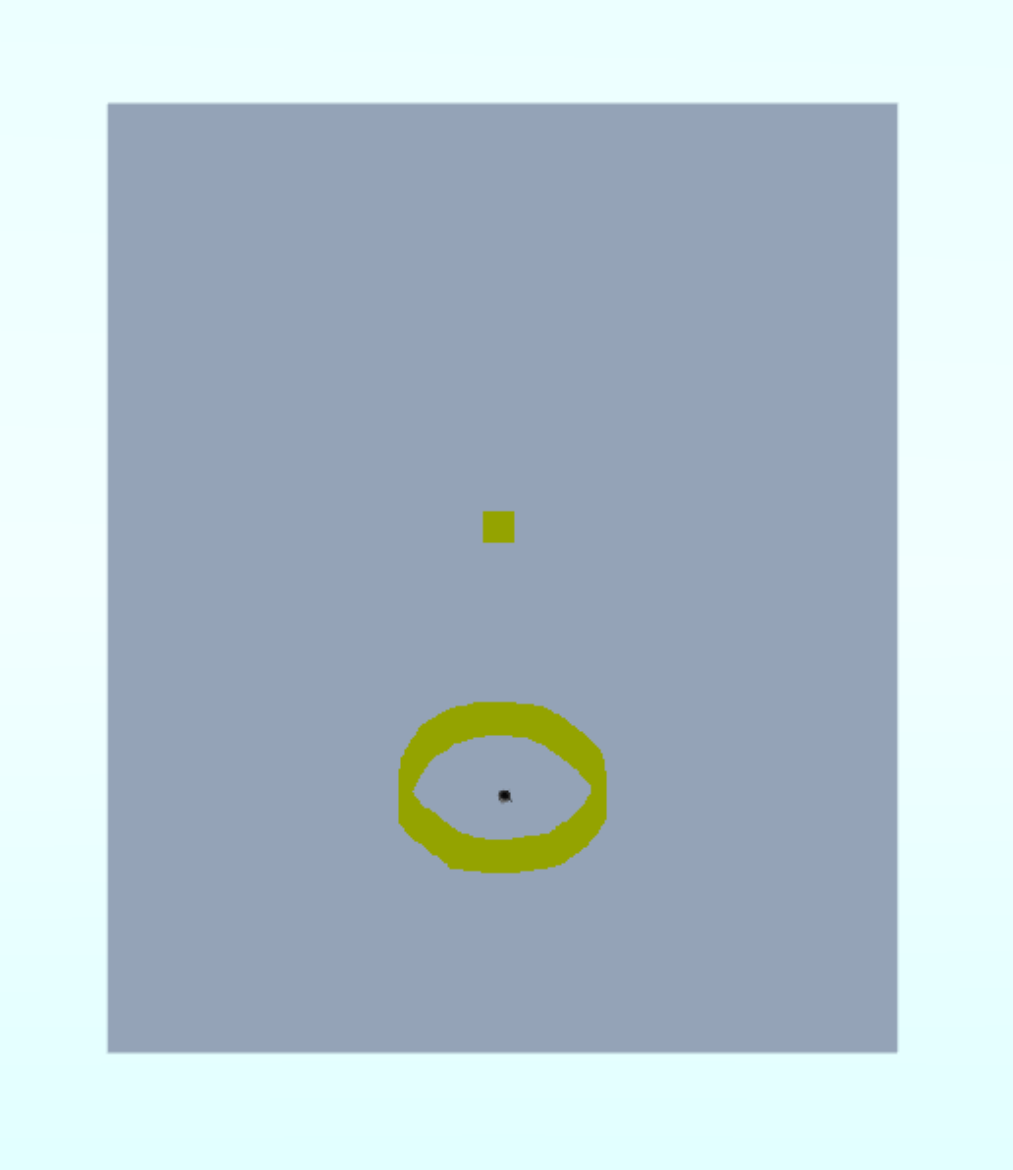
The hit particles have a deactivator script attached to them. So after they are enabled, they disable themselves after a specific time has passed. As soon as they are disabled, they become available to use in the object pool.

Hit Tags

Normally, while writing hit listeners, such as the logic we have written in the previous section, you would want to behave differently according to the hit object's layer or tag. For instance, if it's tagged as "Metal", you'd spawn a metal particle, but if it's tagged "Body" you would spawn a blood particle. The **HitListenerSpawnParticle.cs** script under the folder **InanEvin>Realistic Sniper and Ballistics>src>Hits** is doing exactly this. However, introducing tags such as Metal, Dirt, Body etc. for the demo scenes isn't a good idea, as we want to avoid modifying the project settings of the users as much as possible. That's why, under the same folder you can see scripts such as **HitTagBody.cs, HitTagMetal.cs, HitTagWood.cs**, so instead of checking the hit object's layer or tag, we simply check whether they contain any of these tag MonoBehaviors or not, and spawn particles accordingly in **HitListenerSpawnParticle.cs**. In your own project, you might want to clearly assign reasonable tags to your objects, so you can check their tags instead of checking for component existence.

Zero Distances

A projectile will lose it's velocity and slowly fall towards the ground based on the gravity. In the case of bullets, this is the concept of bullet drop. Let's test this really quickly in our tutorial scene. Take our cube object and position it to **0,0,500.** Then go to our main camera, and set it's **field-of-view** to **1.** This way, we will be zoomed at the cube positioned at 0,0,500. Now play the game and fire, you will realize the bullet drops so much that we are not hitting anywhere close to our center of aim.



Center of aim versus the hit position.

So, we need to aim higher to compensate for the bullet drop. Most modern sniper scopes have the concept of **zeroing.** They set a specific zero distance, which makes the scope

show the target with an angle downwards, causing the shooter to slightly move the weapon with an upwards angle. Thus, the trajectory of the bullet will be such that the bullet will land exactly where we are aiming at, at the zero distance.

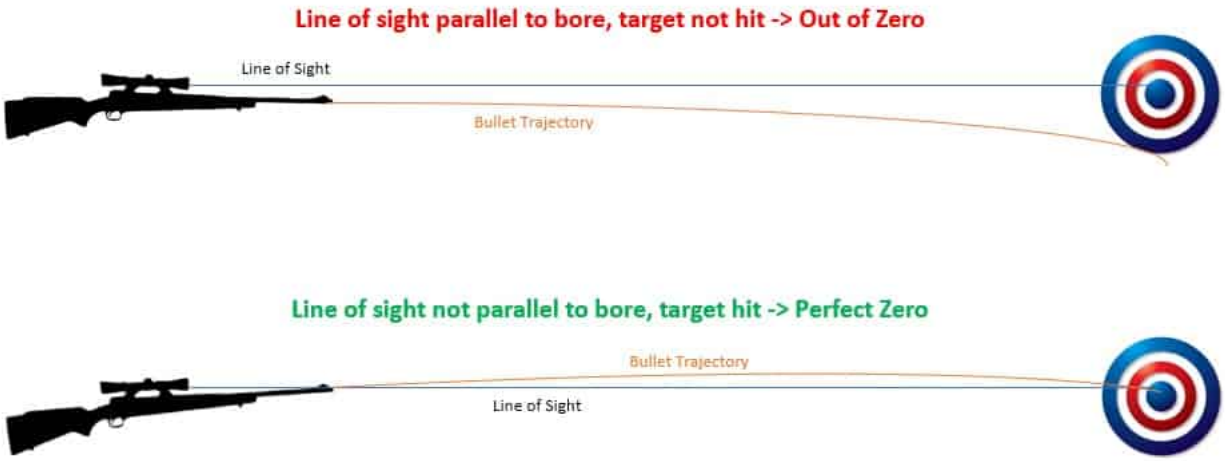


Image from: https://riflescopescenter.com/losing-zero/

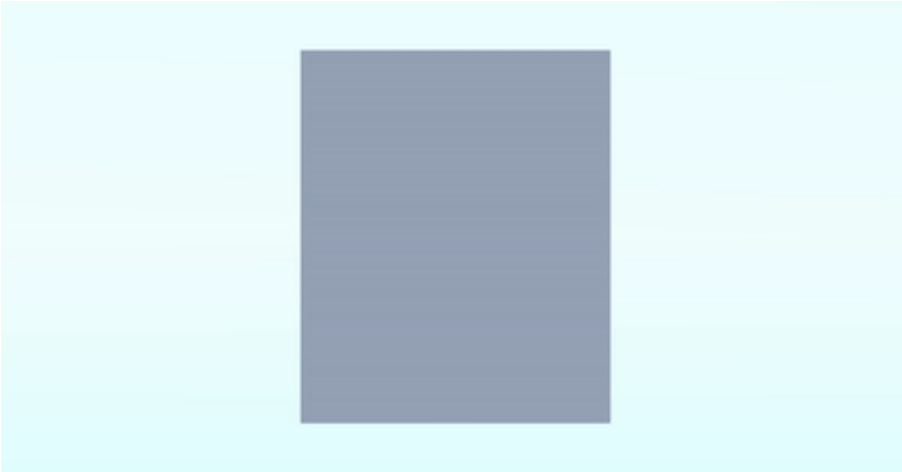
The same concept is implemented in RSB. Remember we had list of available zero distances in SniperAndBallisticsSystem instance? When the game starts, the current zero distance will be set to **0,** indicating no zeroing. But we can change this dynamically by script. Let's upon up our **Tutorial.cs** and write two blocks to check input for changing the zero distance.

RSB-Tutorial20.cs https://gist.github.com/a60f0fbeb5dead8e0578a34d3d42940c.git

We can use the **CycleZeroDistanceUp** and **CycleZeroDistanceDown** methods in SniperAndBallisticsSystem.cs to do this. So let's incorporate them to our if blocks, then also debug the **CurrentZeroDistance** in SniperAndBallisticsSystem.cs instance. So our whole script should look like this:

RSB-Tutorial21.cs https://gist.github.com/016f71c0cc86502e2dce9b6387592f0d.git

Now if you play the game, and cycle the zero distance up to **500** (you can see from the debug console), then fire, you will see that we hit the target exactly at the center.



Target at 500 meters hit at the center with the zero distance of 500 meters.

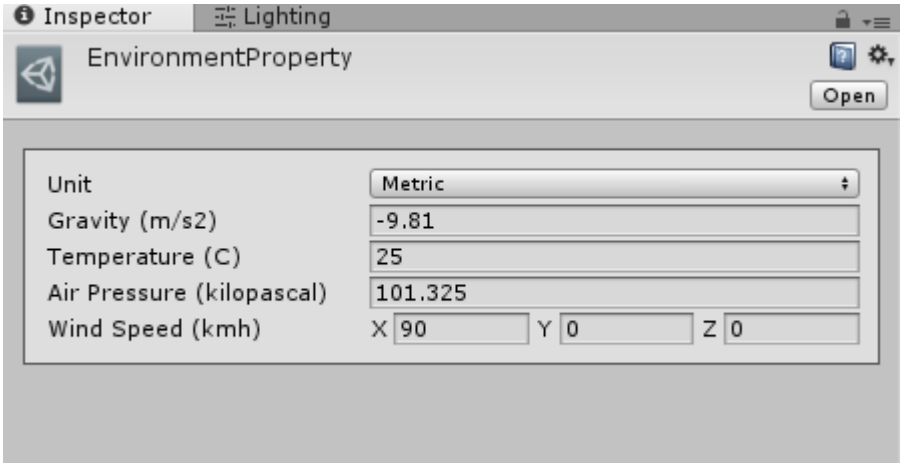
So basically, you can set your available zero distances in SniperAndBallisticsSystem instance, then cycle them using the two public methods provided during runtime. If you are aiming at a target exactly at your current zero distance, you will not see any bullet drop, if you are aiming a target at a further distance, the bullet will still drop **below** your aim center. If you are aiming to the target at a closer distance then the current zero distance, the bullet will hit **above** your aim center.

> It is recommended to have 500 meters as the **maximum** zero distance in the list. Technically, the system can calculate any zero distance up to **2000** meters (limited for performance reasons), but it is not realistic to expect accurate results above 500 meters. A powerful cartridge, such as .388 Magnum or 50 BMG will still result in accurate zeroing around distances such as 1000 meters, but most scopes do not have such accurate calculations. So in most games, 500-750 meters is the realistic upper limit.

Now, let's leave the scene as is, and see how the wind effects the bullet's trajectory.

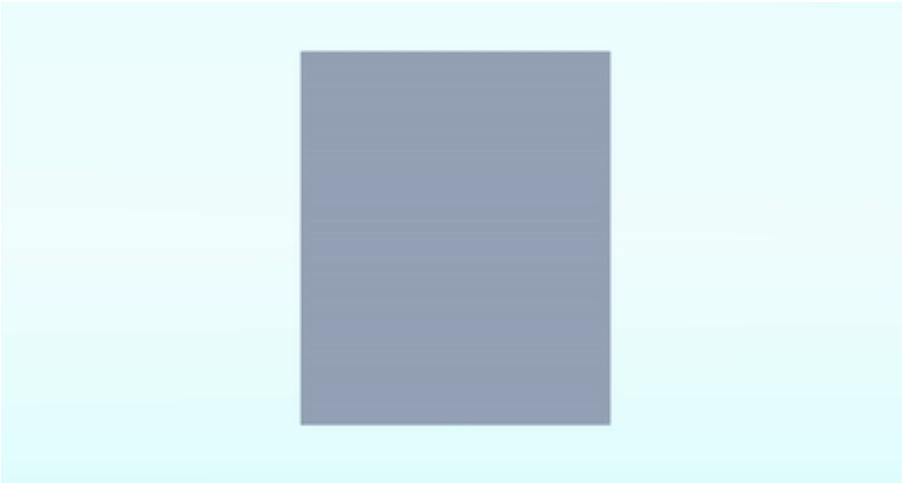
Wind Vector

The wind vector affects how the bullet travels, which can be set in the Environment Properties asset. Select the Environment Properties asset we have created for this tutorial (it is referenced in Sniper and Ballistics object), then change the **wind vector** to **90, 0,0.**



Wind vector set to 90, 0, 0

If we play the game and fire now, we will see the bullet landing to the right of our aim center.



Bullet hitting to the right of our aim center, also dropping.

If we set the zero distance to **500** as shown in the previous section, we will not see any bullet drop, but the wind will still affect the bullet. It is possible to dynamically change the wind vector by reaching to the **SniperAndBallisticsSystem.instance.GlobalEnvironmentProperties.WindVector** value. An

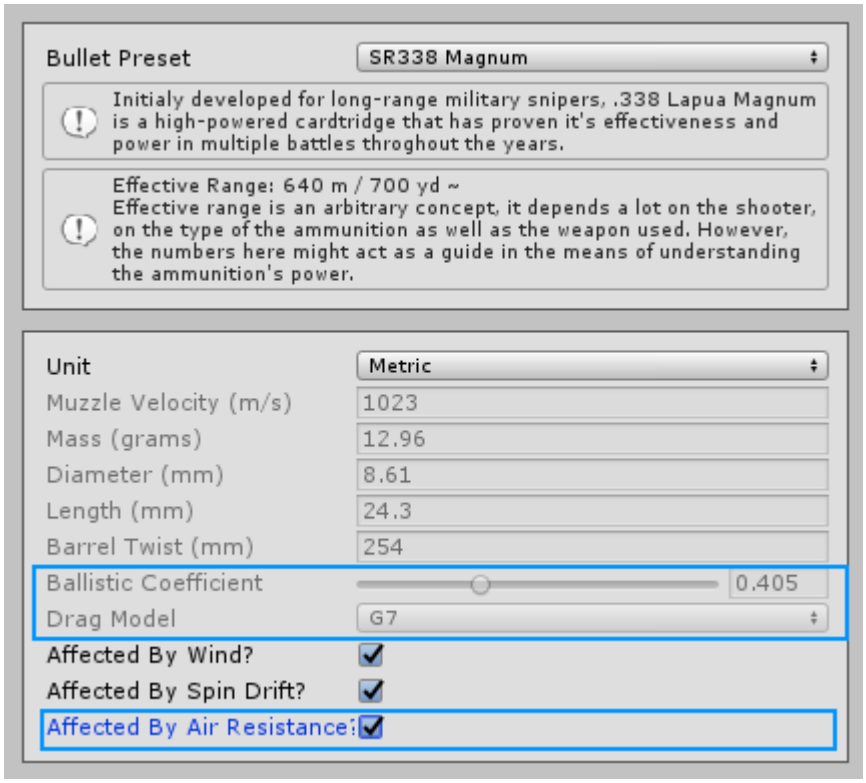
example of this can be found under the folder **InanEvin>Realistic Sniper and Ballistics>src>Utility,** see the **WindRandomizer.cs** script.

Setting zero distances compensates for the bullet drop, and a scope system shall compensate for the wind vector. Ideally, we can write a scope system that will show us where the bullet will land depending on the current wind vector. The Dynamic Scope System in RSB does exactly this, which will be explained in the later sections.

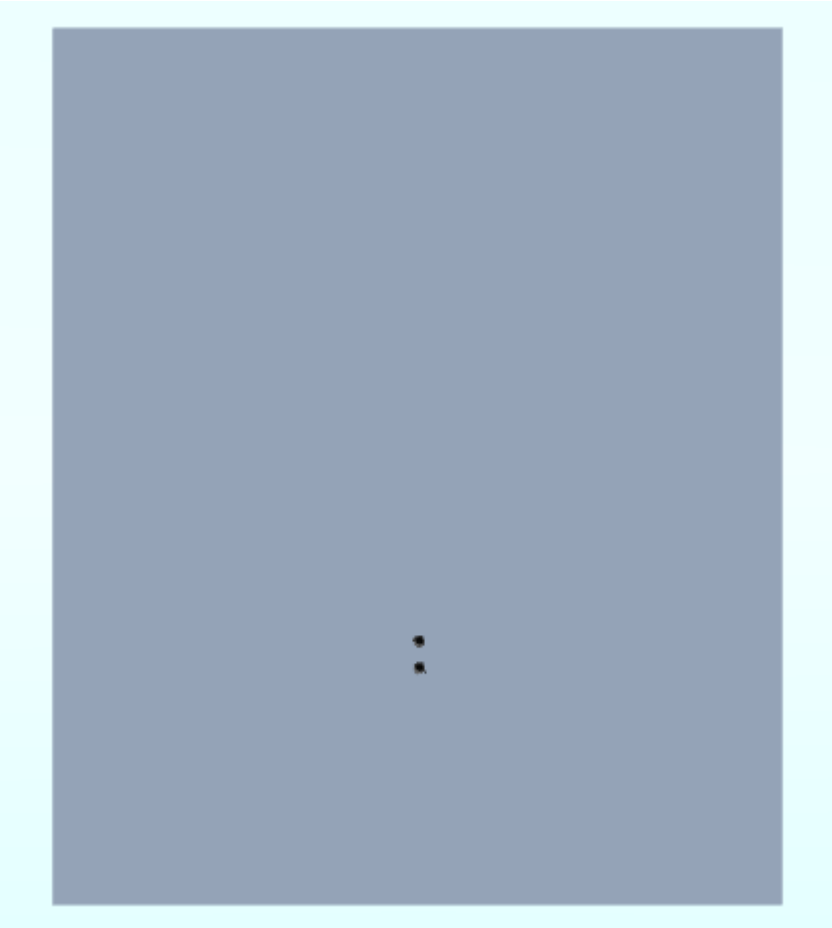
> ⚠ Let's **reset** the wind vector to **0,0,0** before continuing the sections.

Air Resistance

On top of the gravity and the wind, the bullet's are also affected by **Air Resistance.** The **Drag Model** and **Ballistic Coefficient** in **Bullet Properties** asset determine the bullet's behavior based on air drag.



Higher the **Ballistic Coefficient**, easier the bullet will punch through the air, causing it less resistance. If we are to select our Bullet Property asset we use for this tutorial, disable **Air Resistance** by unchecking **Affected By Air Resistance,** then fire, we would see that the bullet drops less.



Bullet hit without air resistance (above), bullet hit with air resistance(below)

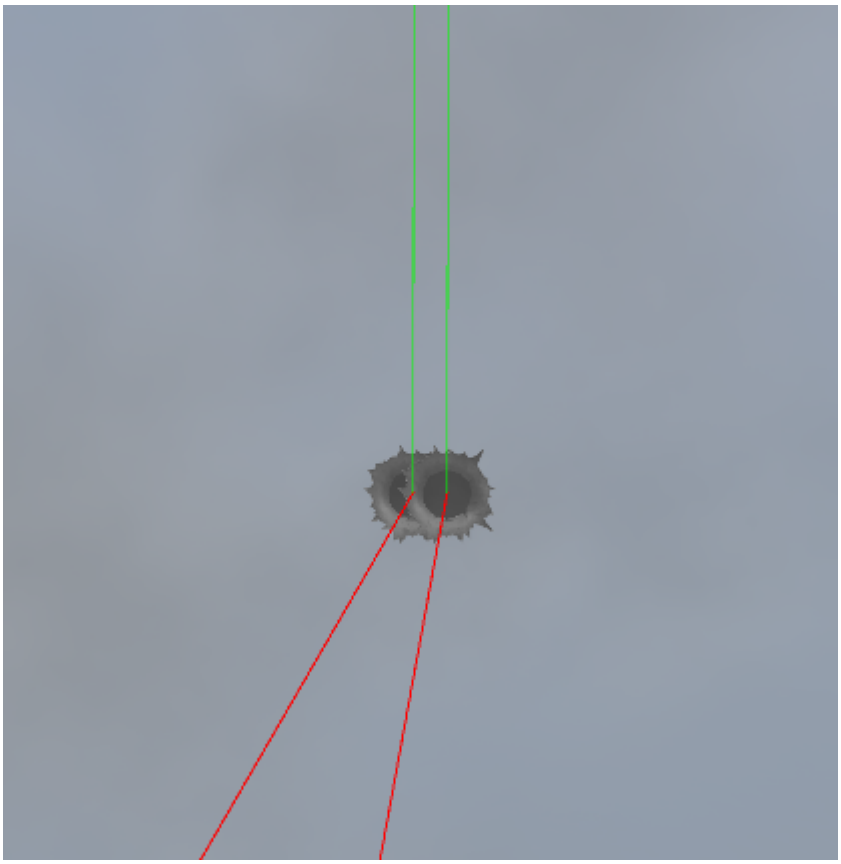
This is because the air resistance causes the bullet to slow down. Slower the bullet, more time it takes for it to reach to target. More time it spends on air, more drop occurs due to gravity. You can easily tweak this behavior on the Bullet Property asset to suit to your needs, but it's recommended to leave the **air resistance on.**

Less powerful cartridges such as 9mm are affected much more by air resistance.

Let's **enable back** the **air resistance** by checking **Affected By Air Resistance** checkmark in our Bullet Property that we use for the tutorial before continuing the next sections.

Spin Drift

When the bullet leaves the barrel, due to bullet's spin inside the barrel caused by the barrel's twist, the bullet will have a little tendency to go towards the right/left of the barrel. In most cases, the bullet goes towards the right, since most barrels are twisted this way. This results in the bullet to land over to the right of the aim center. However, this effect is extremely minimal, only couple of centimeters in hundreds of meters of distances. In RSB, this is simulated as well, by checking or unchecking **Affected by Spin Drift** checkmark, you can determine whether this will be used or not.



Spin drift effect in 1000 meters.

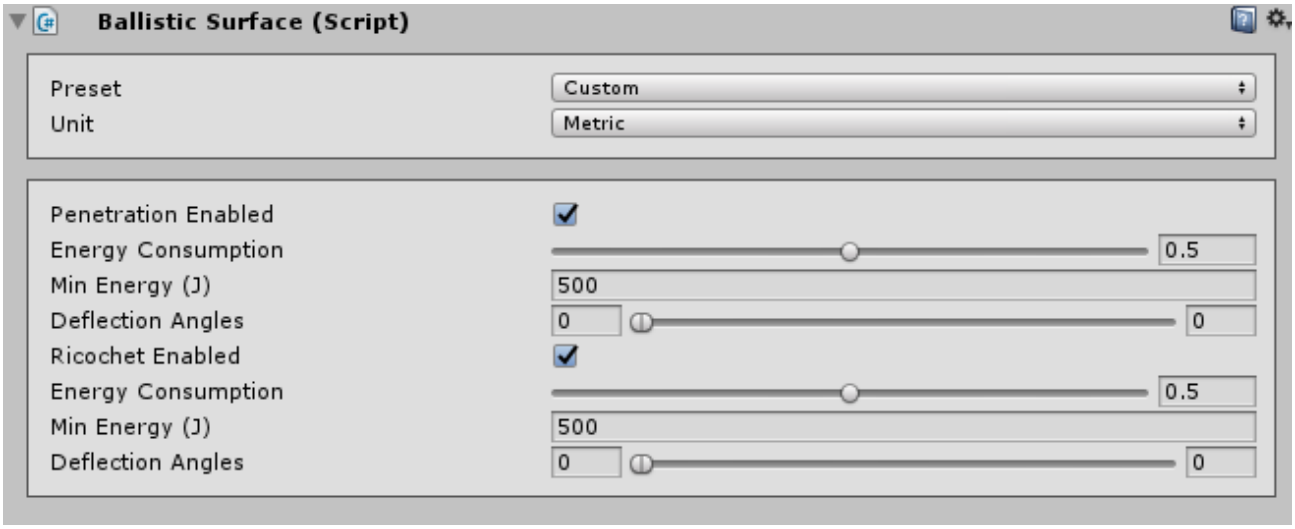
Less powerful cartridges such as 9mm are affected much more by spin drift.

Let's **enable** spin drift on our Bullet Property used for this tutorial before continuing the next sections.

Ballistic Surfaces - Penetration

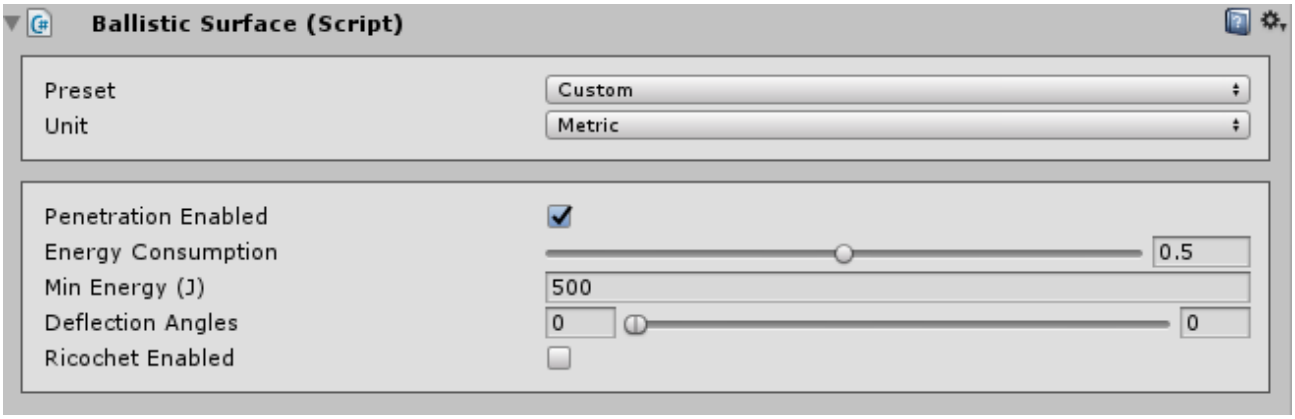
> We have modified our cube's position in previous tutorials, so let's put it back to **0,2,10** before continuing here. Also, make sure our **main camera**'s **field of view** is set back to **60.**

When you want any of the objects in your scene to be penetrable by bullets, you need to attach **BallisticSurface.cs** script onto your objects. Let's add this component to our cube object in the scene.

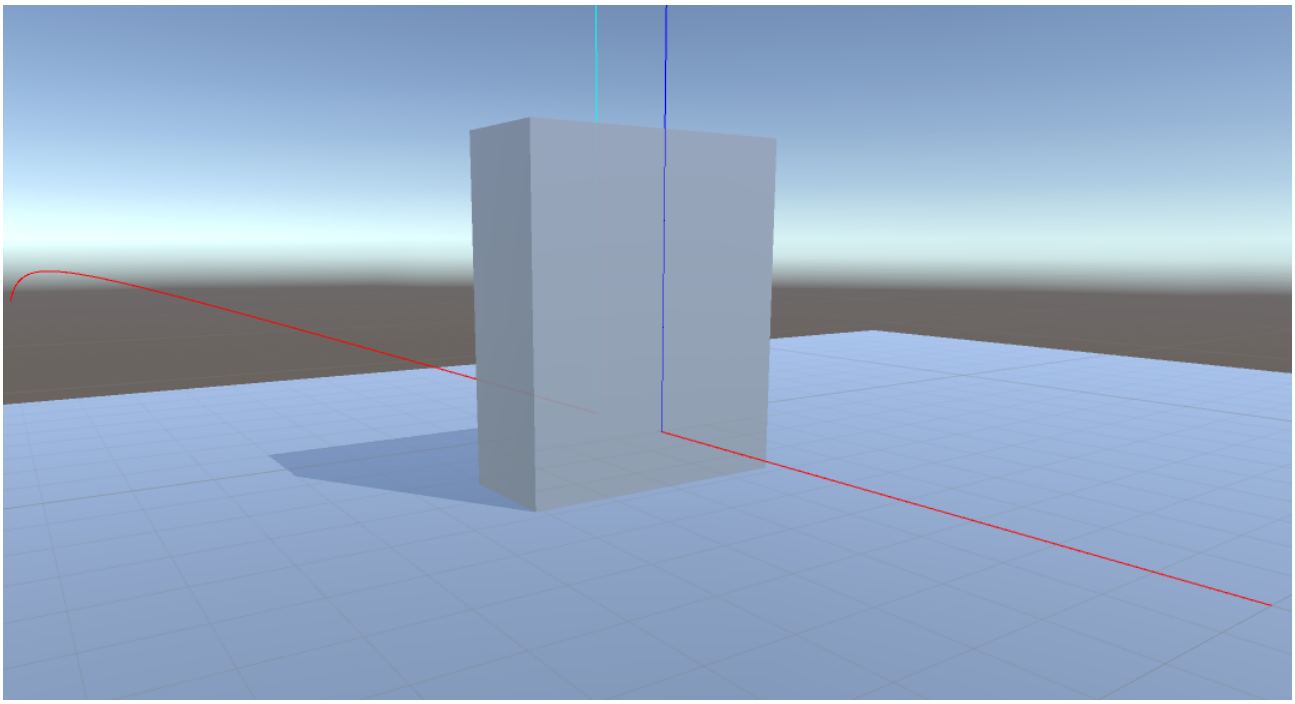


BallisticSurface.cs component on the cube object.

You can use the **Preset** drop-down to select various different surface presets. For the sake of the tutorial, we will use the **Custom** preset, and adjust the variables ourselves. For this tutorial, we are only interested in penetration, so go ahead an uncheck the **Ricochet Enabled** checkmark. When a bullet starts penetrating a surface, it will lose it's kinetic energy depending on the **Energy Consumption** slider. Here, a value of 1.0 means losing 100% of the bullet's kinetic energy, while 0.0 means losing none. **Min Energy** denotes the minimum kinetic energy a bullet should have in order to be able to start penetrating this object. **Deflection Angles** two-ended slider allows us to define an angle range, which will be used to select a random exit angle for the bullet. Let's leave the settings like this for now, except that we have unchecked **Ricochet Enabled** mark, so it will look like this:

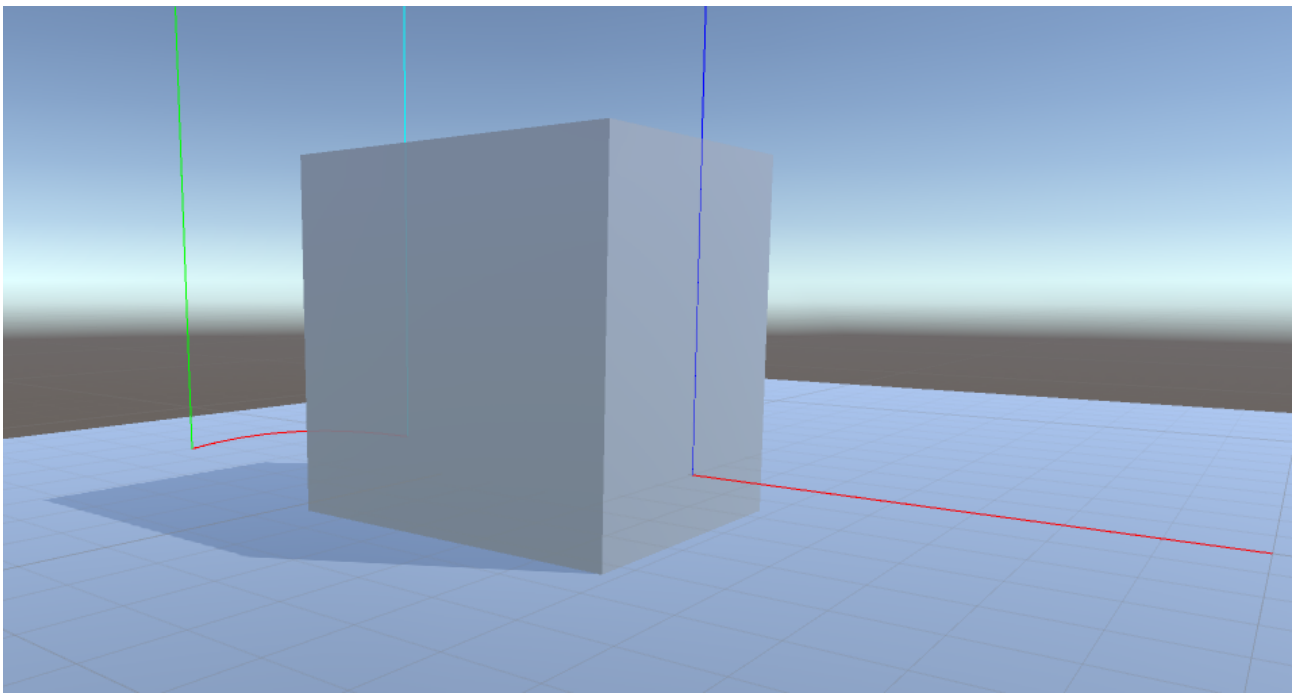


If you play the game and hit space now, you will see that the bullet will penetrate the object.



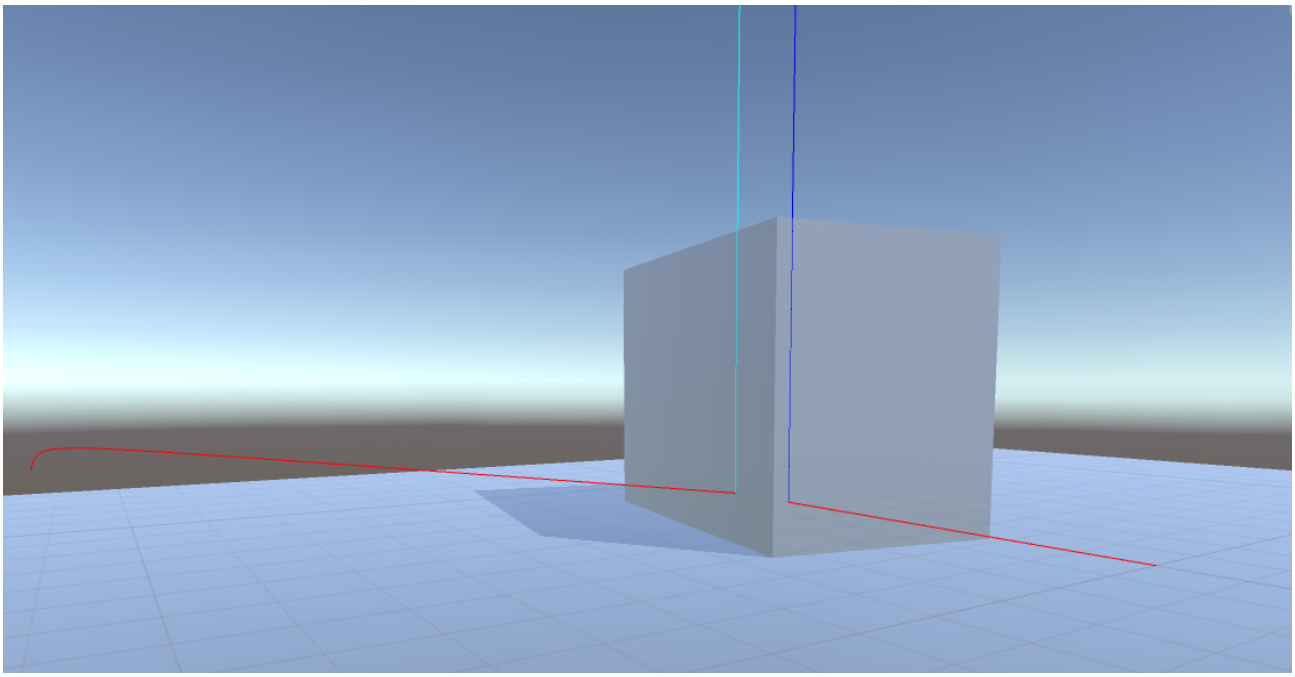
Bullet raycast penetrating the object. Blue line shows a PenetrationInHit, meanwhile cyan line shows a PenetrationOutHit.

More the bullet ray stays inside the object, more kinetic energy it will lose, depending on the **Energy Consumption**. So, object's thickness will affect how the bullet penetration continues. Go ahead and change our cube's scale to 6 in Z axis, so it's scale will be 5,6,6. Then play the game, hit space bar. You will see that the bullet still penetrates, but loses so much of it's kinetic energy that it basically falls into the ground right after exiting the object.



Bullet ray still penetrating when the object is thicker, but falling right after to the ground (green hit line).

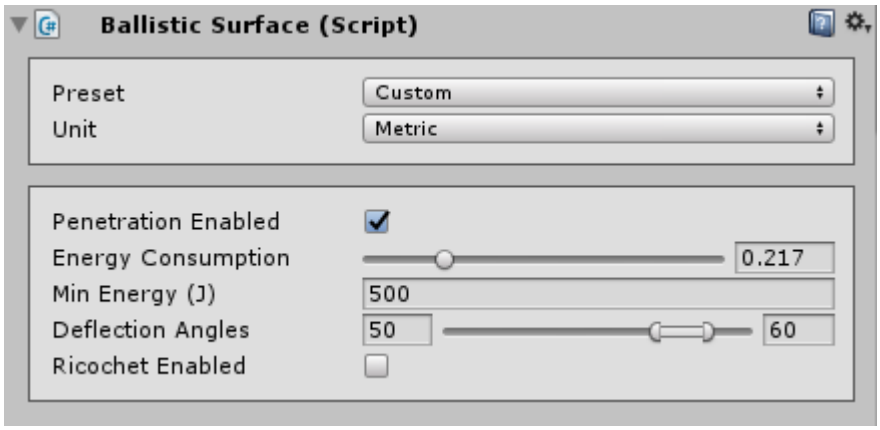
If you were to further increase Z axis scale, say to 8, then the bullet ray would not get out of the object. Also, the thickness doesn't have anything to with the actual Z scale of the object. So, say you set the Z scale to 8, preventing the bullet from penetrating. But if you moved the cube to such a position that the bullet should enter from not-so-thick part of the cube, it would still penetrate.



Still penetrating because ray hits such an angle that the it could get out before losing too

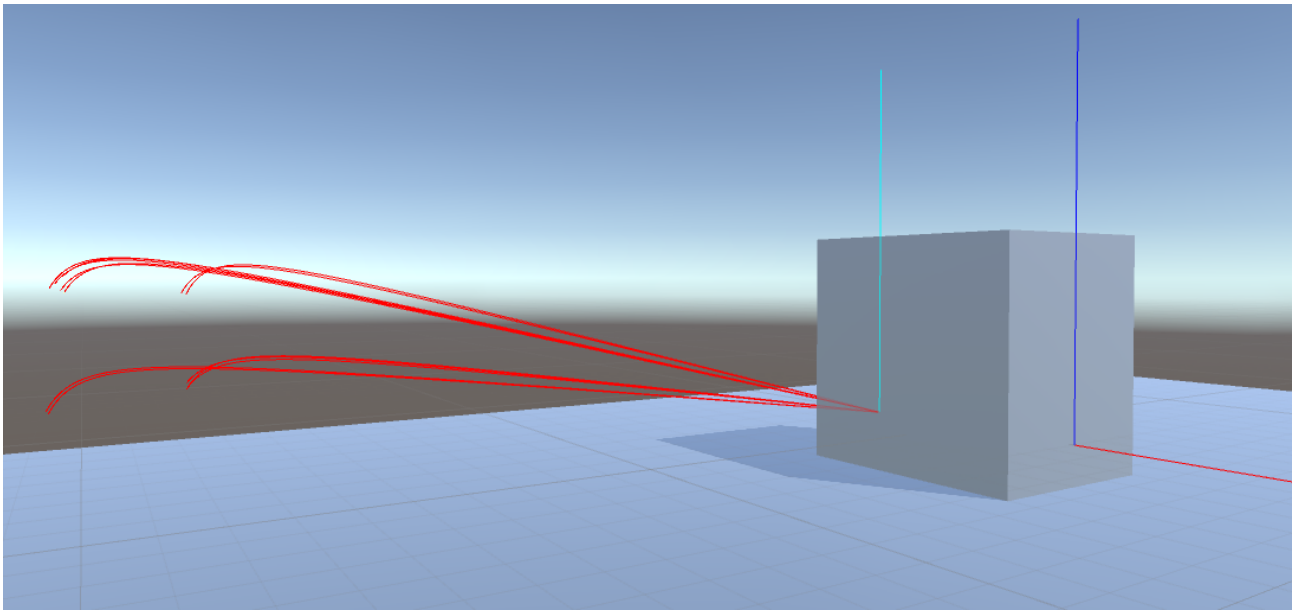
much energy.

Of course you can tweak this behavior completely to your liking by changing the **Energy Consumption** value. Finally, let's change the **Deflection Angle** value, to be somewhere between 50 and 60.



Deflection angles set to 50-60 range.

Now if we were to fire multiple shots continuously, we would see that they would be deflecting from the linear path randomly according to the deflection angles.

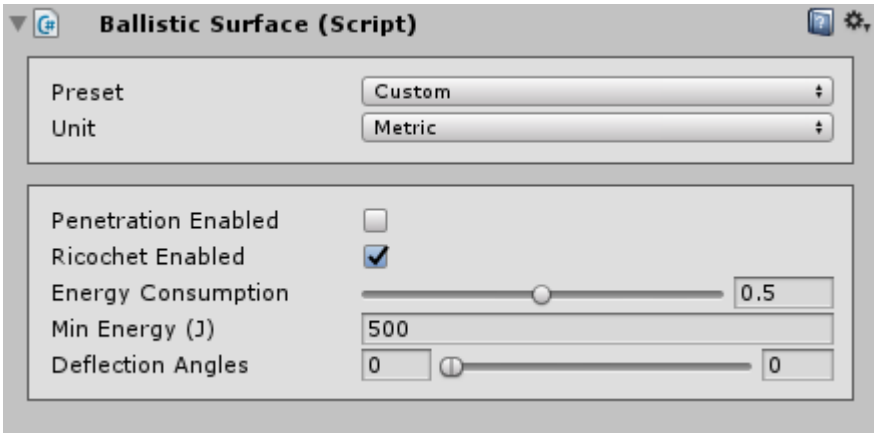


Multiple shots fired continuously, after penetrating the rays are deflected.

Now let's take a look at the **Ricochet** settings to see how the bullet rays would ricochet off the surface.

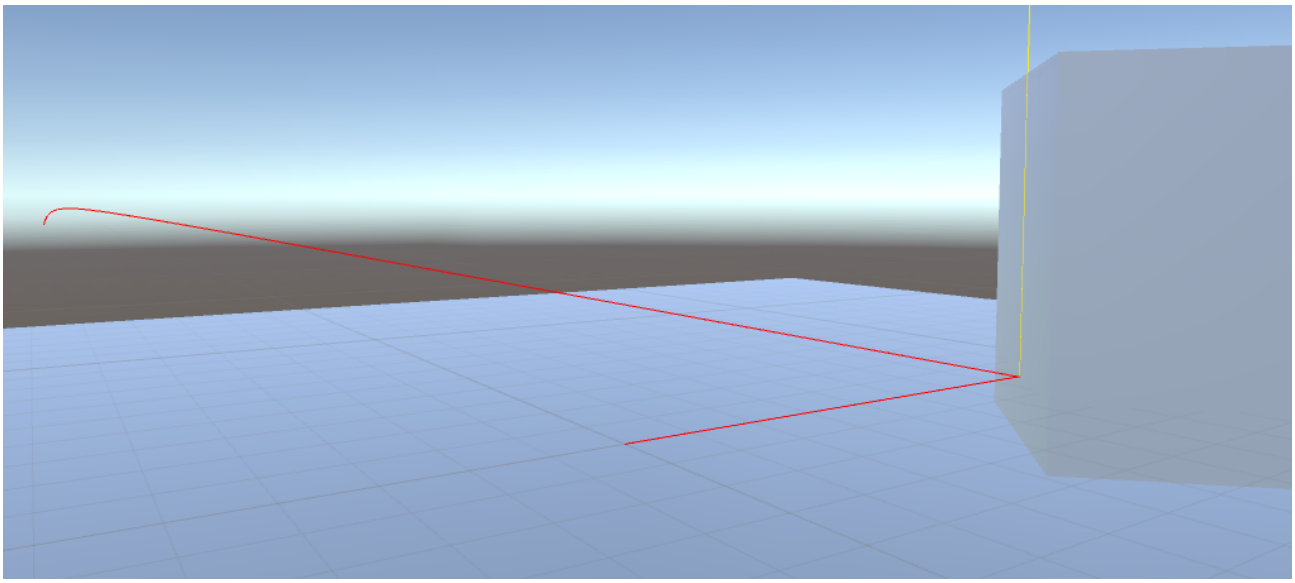
Ballistic Surfaces - Ricochet

Ricochet feature is extremely similar to the penetration feature. First, let's disable the penetration, but enable ricochet on our cube object.



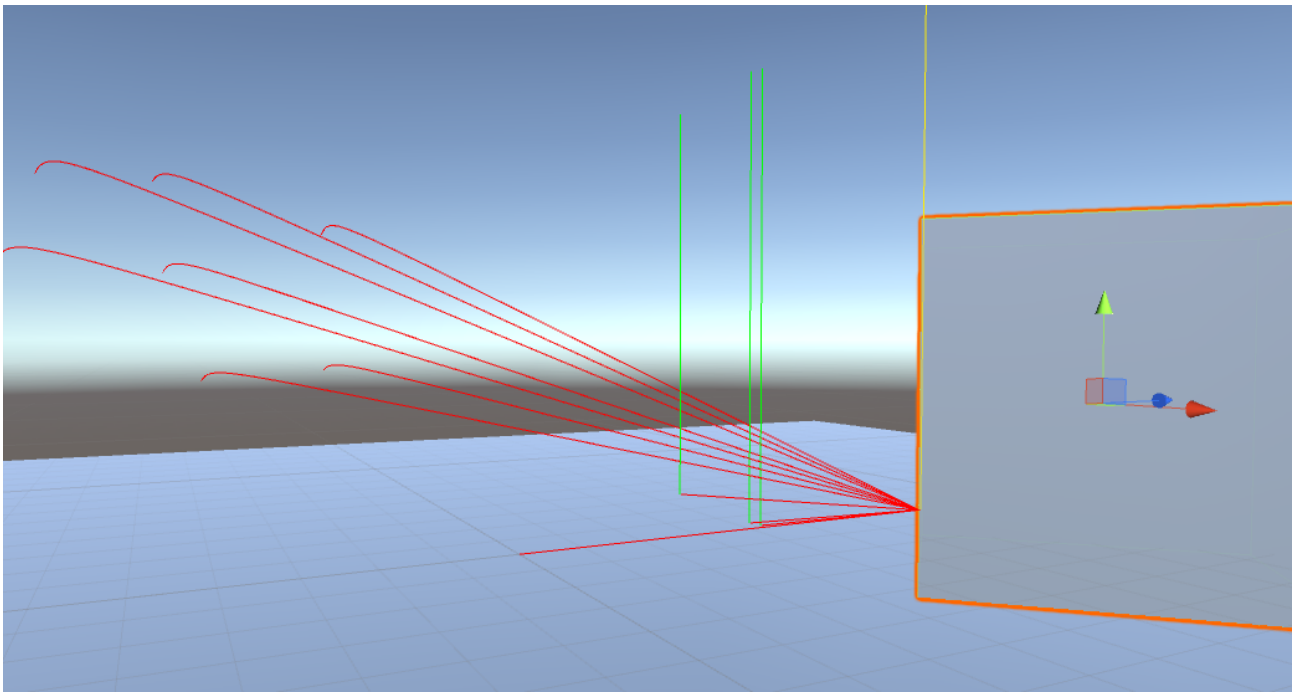
Ricochet settings enabled.

The same principles apply as the penetration settings. The ricocheting bullet will lose it's kinetic energy based on the **Energy Consumption**. However, unlike penetration, the bullet will lose the energy only once, when it hits the object. So object's thickness play no role in here. Now, let's move our cube in X axis by 2 units, so it's position will be 2,2,10. Then rotate the cube on Y axis by 45 degrees so it's rotation will be 0,45,0. If you take a shot now, you will see the bullet ricocheting off the surface.



Bullet ray ricocheting off the surface of the cube.

By changing **Deflection Angles** to a range between **10-20**, we will get randomly deflecting bullet rays after ricocheting off the surface.



Rays ricocheting off the surface, some of them even hit the ground.

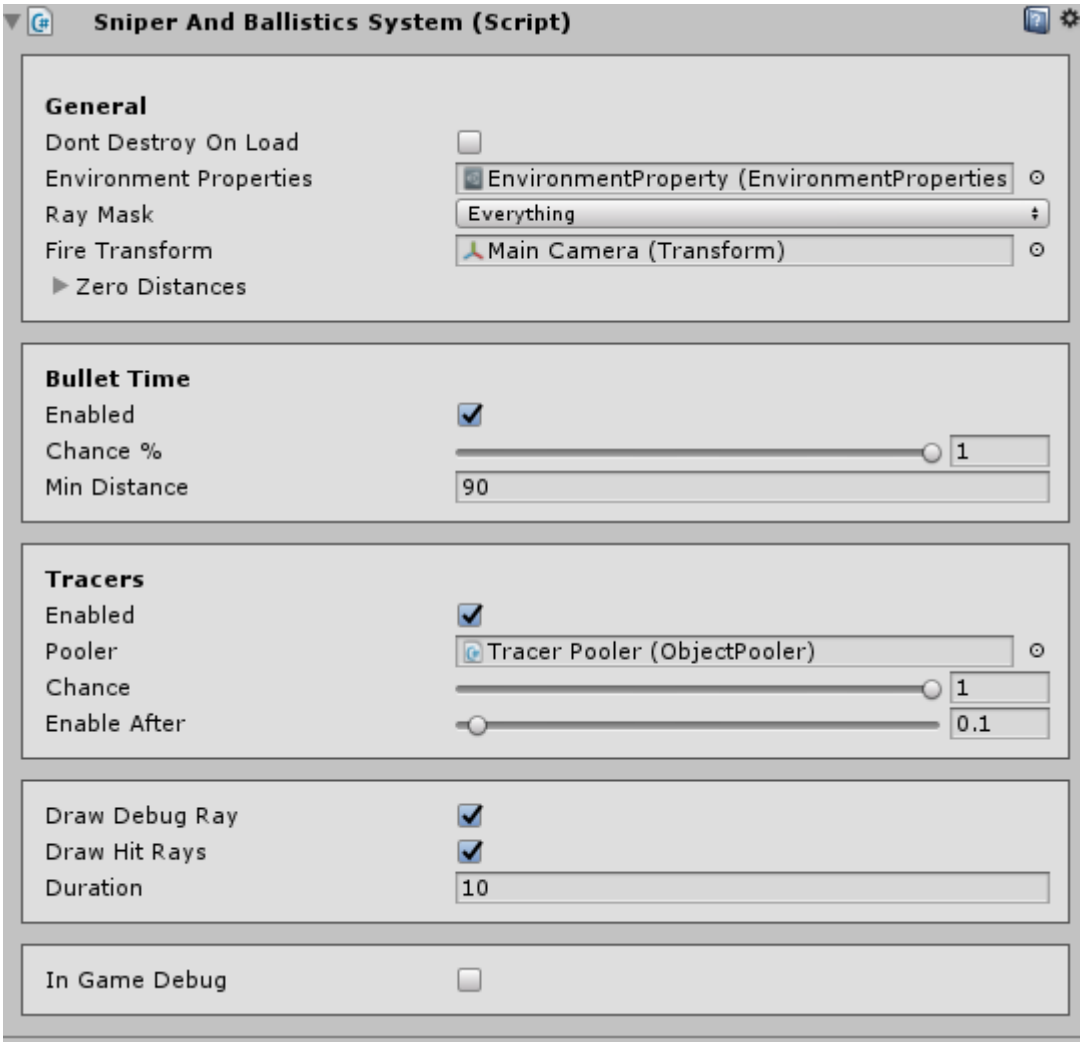
By playing with the values, you can create a penetrable/ricochet-able surface of your liking. You can take a look at the presets included to learn how to simulate some common surfaces.

> If you want to use both penetration and ricochet at the same time on the same object, you should know that the system always favors penetration over ricochet. So a bullet ray hitting a surface, will first check if the object will have **BallisticSurface.cs** script on it. If so, it will **FIRST** check whether the penetration is enabled or not, and whether the current energy values allow penetrating the object. If that fails, **ONLY THEN** the ricochet settings are checked.

Bullet Time Effects

Bullet Time effects are small cut-scene like sequences showing the bullet flying towards it's target. It became popular in game franchises such as Sniper Elite and Sniper Ghost Warriors. RSB provides a powerful tool to create your own bullet time camera behavior without writing a single line of code. Alternatively, you can use RSB's powerful event system to write your own camera logic as well. First, let's take a look at how we can incorporate this powerful tool into our tutorial scene.

We first need to enable bullet time effect on **SniperAndBallisticsSystem.cs** instance in our scene. Navigate to the Sniper and Ballistics object containing the script, and enable bullet time checkmark.



Bullet time effects enabled in SniperAndBallisticsSystem.cs instance.

Chance determines the percentage which the bullet time effect will get triggered on each time a **bullet time target** is hit. **Min Distance** determines the minimum shot distance for the effect to get triggered. Let's leave the values as is for now.

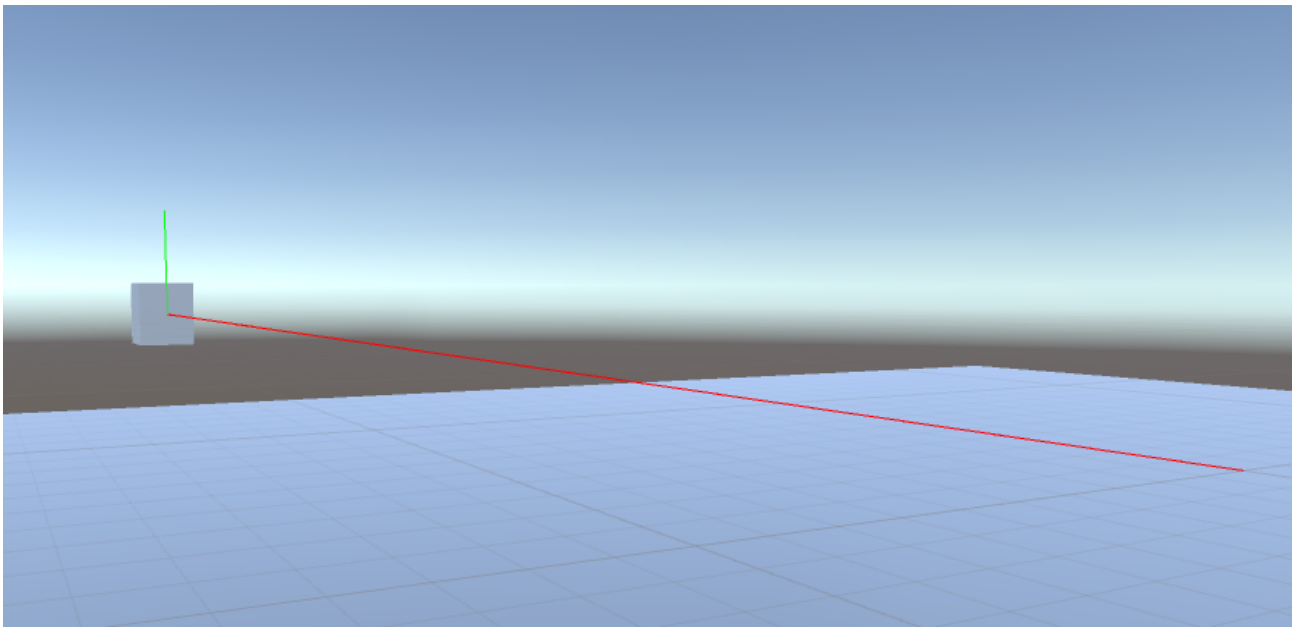
Target

In order for RSB to trigger a bullet time effect, it needs to hit a target that can be, let's say bullet-time-able. This is determined by the **BulletTimeTarget.cs** script, which can be found under **InanEvin>Realistic Sniper and Ballistics>src>BulletTime** folder. Let's attach this script to our cube object

BulletTimeTarget.cs attached to our cube object.

> Note that we have removed the **BallisticSurface.cs** script from the cube for the sake of the tutorial, however it **doesn't** affect the bullet time at all. So an object can be penetrable, or ricochet-able, and **still** can be a bullet time target. RSB is only concerned with wheter the **BulletTimeTarget.cs** script is attached or not.

Now let's move the cube to **0,0,100** position, so it's far away from the camera. This way, the default minimum distance of 90 will not prevent the effect from being triggered. Also, let's scale our cube to **5,6,1** and make sure it's rotation is **0,0,0**. If you fire at this point, there won't be any changes in the scene when the target is hit.

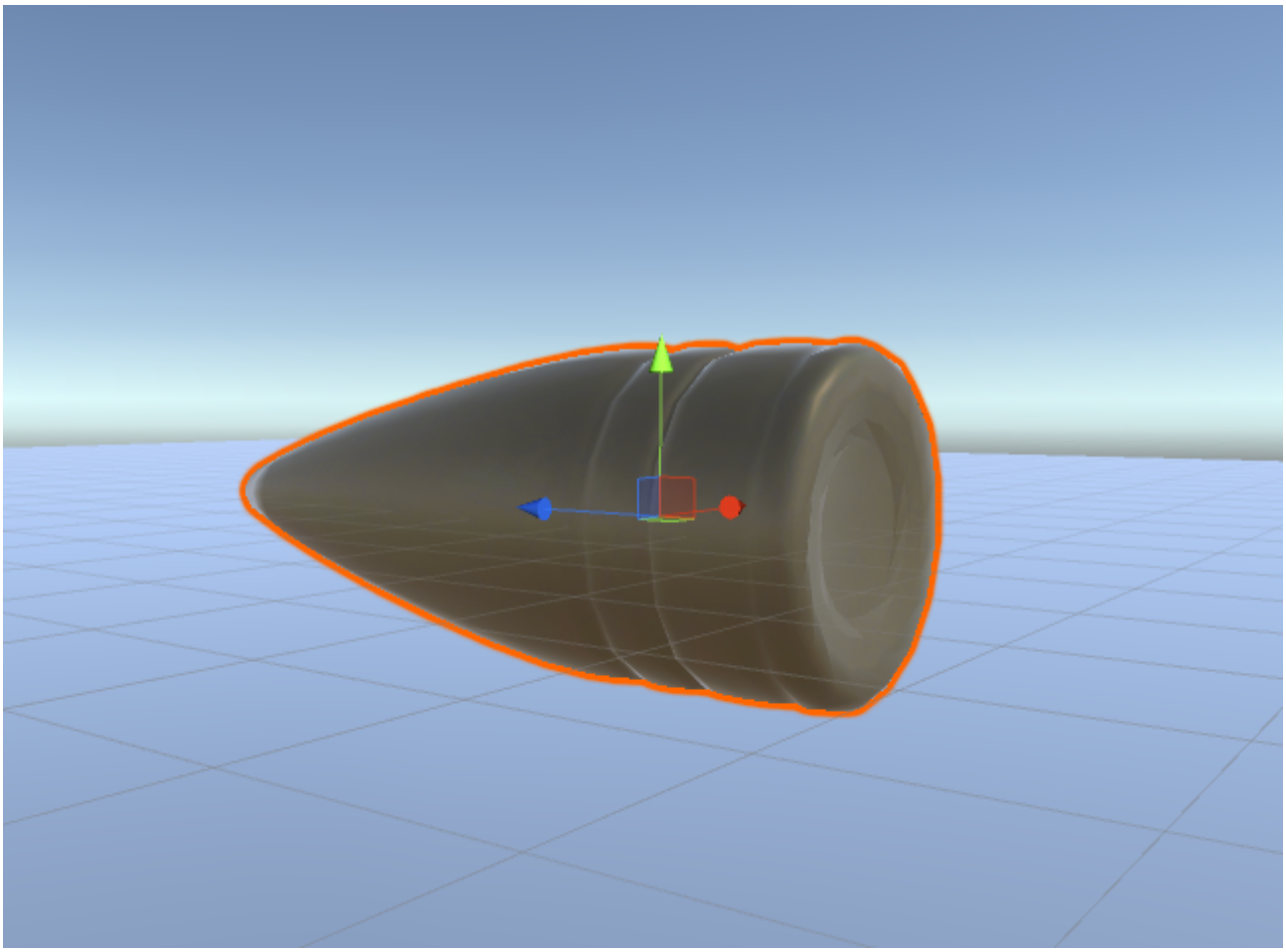


Hitting the cube at 0,0,100 which has BulletTimeTarget.cs on it. No effect is triggered at this stage.

Right now, the bullet time effects are not being triggered, because when we are firing using **FireBallisticsBullet** method, we do not pass in any arguments for bullet time. Before doing so, we would need to setup a bullet mesh in the scene which will be visible during bullet time effects.

Bullet

Whenever you trigger a bullet time effect, there needs to be an actual bullet object travelling throughout the scene to watch. We can use the already available bullet prefab for now. Navigate to the **InanEvin>Realistic Sniper and Ballistics>System Prefabs** folder and drag & drop the **Bullet Time - Bullet** prefab into the scene. If you enable and look at it, it's just a simple bullet mesh, with a rotator script on it. We keep it disabled in the beginning because we don't want it to render except in those times where bullet time is triggered. We will send this object to the SniperAndBallisticsSystem.cs instance while we are calling a method to fire, and it will enable it when using bullet time effects.



Bullet Time - Bullet prefab. Later on, you can replace this with your own bullet object.

Firing with Bullet Time

Open up our **Tutorial.cs** script. We need to create a reference to the transform of the bullet object, as well as to the main camera.

RSB-Tutorial10.cs https://gist.github.com/a7a588f0ddc5018ef2f835475648278f.git

Then we will need to send this object to the system when we are requesting to fire. Remember the **FireBallisticsBullet** method. The method actually can take two more parameters which are null by default:

- **bulletTimeTransform:** This is the transform where the bullet will come out of during bullet time effects (e.g an empty object at the end of the weapon barrel). For now, we will use the main camera transform for it.
- **bulletTimeBullet:** The transform of the bullet object that will be moving towards the target during bullet time.

So, now let's update our call by incorporating these two values in it.

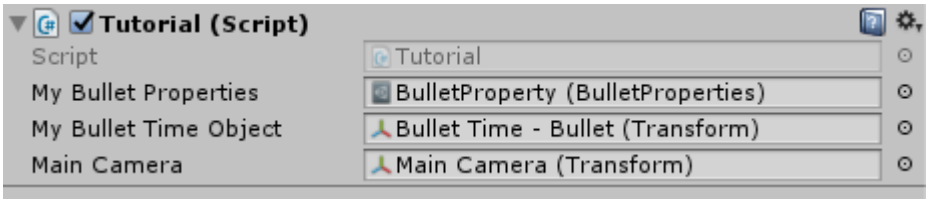
RSB-Tutorial11.cs https://gist.github.com/55181f13f292a9c4b6ef5fee6e85b37d.git

So, our whole **Tutorial.cs** script now looks like this:

RSB-Tutorial12.cs https://gist.github.com/ec8cc7ec85243ecd5913c196c6675e7f.git

Updated Tutorial.cs script, now we use mainCamera & bulletTimeBullet objects in our Fire call.

Do not forget to assign the transforms in the inspector of Tutorial object.

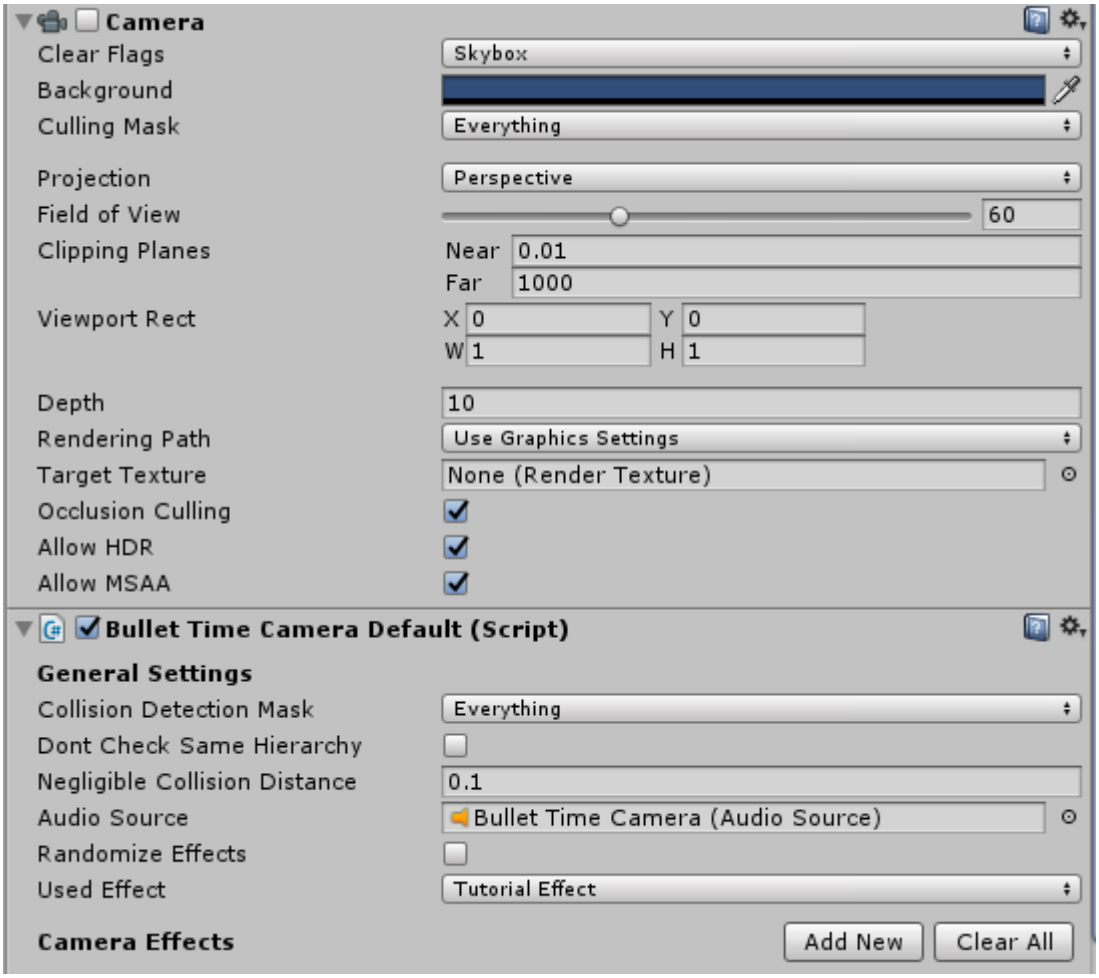


Transforms assigned in Tutorial.cs

If we have fired now, again nothing would change in the scene. This is because even though RSB is now ready to fire up bullet time events, there is no listener in the scene to react to those events. Thats where **BulletTimeCameraDefault** script comes in play, now let's take a look at that.

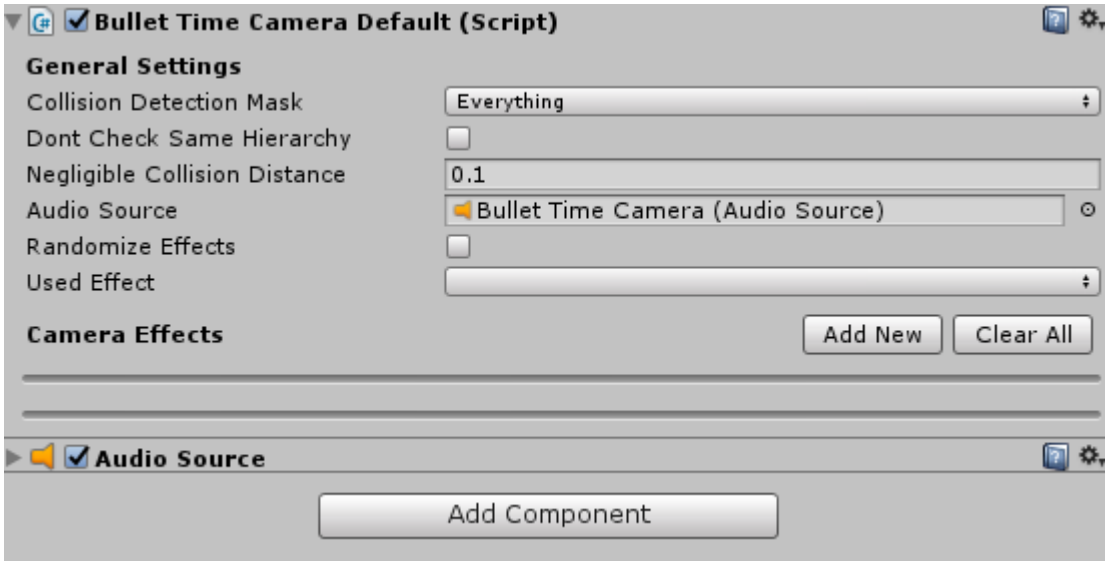
Bullet Time Camera - Default

In order for bullet time effects to be triggered, there needs to be an object in the scene that listens to these events. The logic of creating bullet time in RSB is that there should be an inactive camera in the scene with a script attached to it that enables the camera when bullet time is triggered. The same script shall be responsible for moving the camera to watch the bullet object. RSB already provides this logic in an extremely intuitive and powerful manner. Let's create a new empty game object, rename it to Bullet Time Camera. Then add the **BulletTimeCameraDefault.cs** script on it, which can be found under **InanEvin>Realistic Sniper and Ballistics>src>BulletTime** folder. This will automatically add a camera component to the object as well. Set the depth of the camera component to a large value, such as 10 and set the near plane to 0.01, then disable the camera.



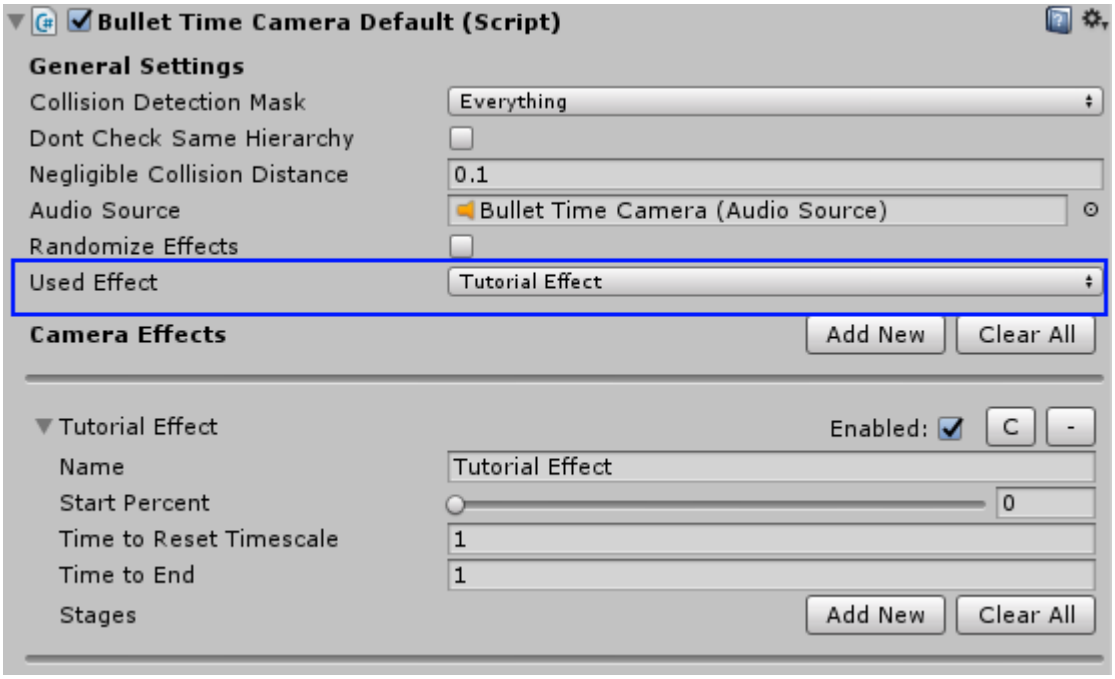
BulletTimeCameraDefault.cs added to the object, camera is disabled, near plane is 0.01 & depth is 10.

The first settings you see are about collision avoidance. This camera will try to avoid objects getting in the way between the camera object and the bullet object. Set the **Collision Detection Mask** to **Everything** for now. Then, also add an **Audio Source** component to the same game object, and assign to the **Audio Source** field. Any audio clips played by the effects will be played on this component.



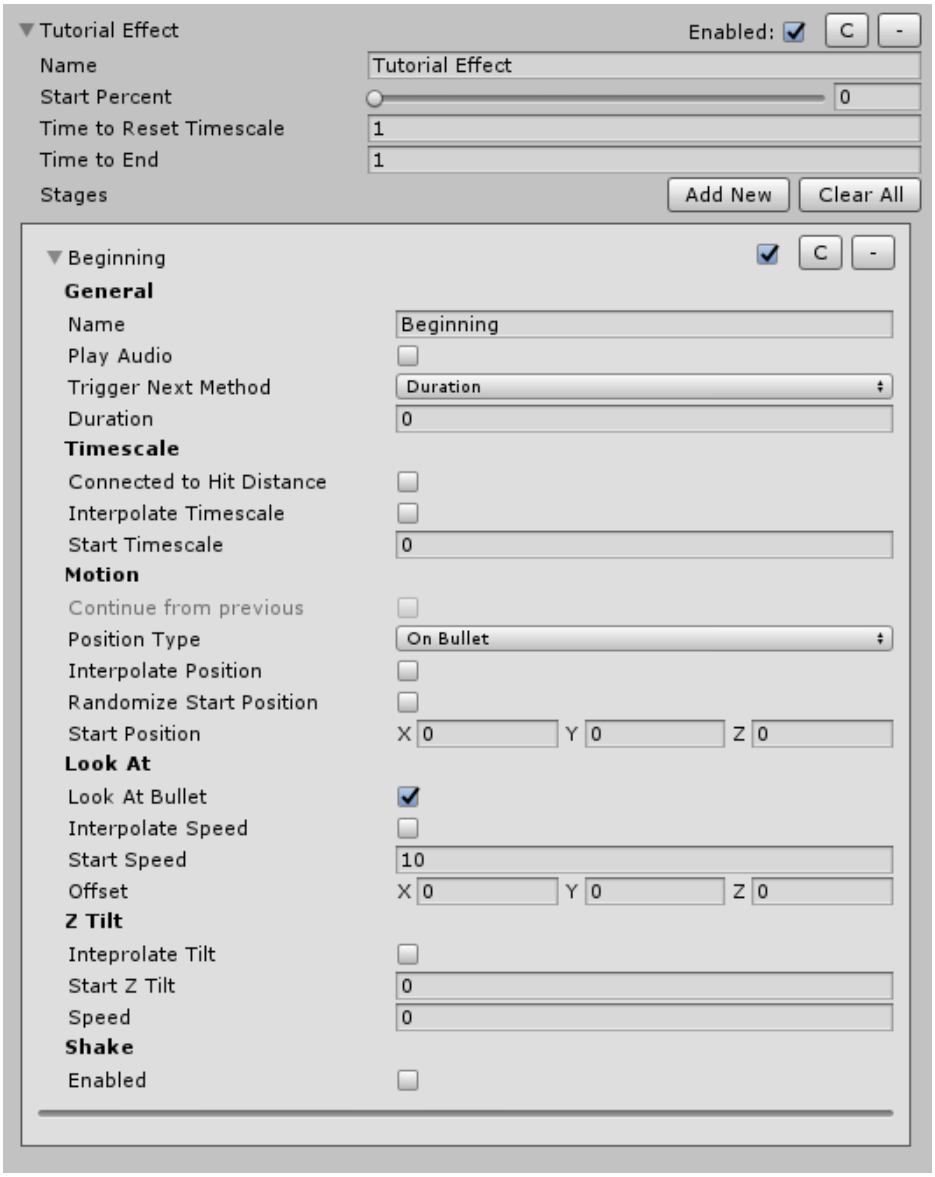
Collision Detection Mask set to Everything, Audio Source referenced.

Bullet Time Camera Default tool works on **effects** and **stages.** Whenever a bullet time event is triggered, an effect will be selected, either by you from the inspector, or as random. Then, the system will play that effect's **stages** in order. So first, let's create a new effect by hitting the **Add New** button in **Camera Effects** field. Open up the new effect's foldout and rename it to **Tutorial Effect.** Then select this effect from the **Used Effect** dropdown.



Tutorial Effect added & selected as the used effect.

Now if we play and shoot, nothing will happen, as this new effect does not contain any **stage**. So let's create a new stage, by hitting the **Add New** button over the **Stages** field. Then open the stage foldout, rename it to **Beginning.**

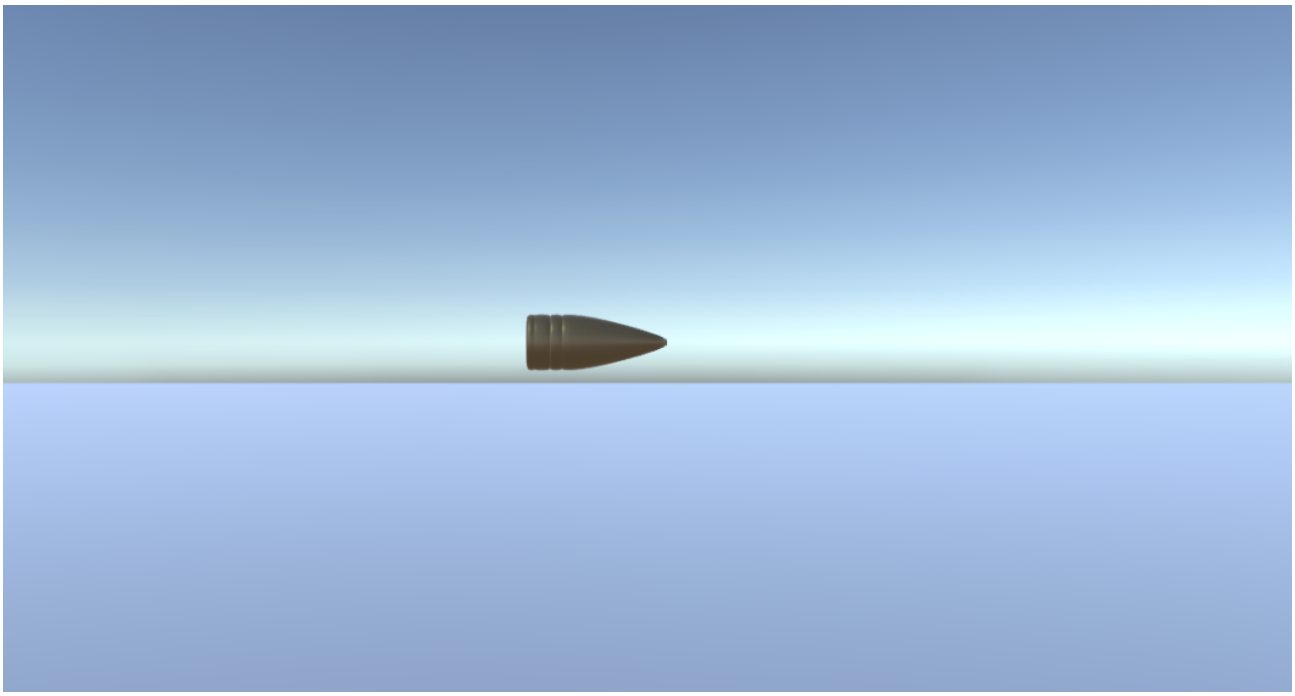


A new stage is added & renamed as Beginning.

Each stage has the power to manipulate different properties of camera, which are:

- **General:** Determines whether this stage will play an audio and how will the next stage get triggered.
- **Timescale:** Determines the timescale of the bullet time when this stage is triggered, whether it will be interpolated or not.
- **Motion:** Determines how the camera will be placed, whether it's position will be interpolated or not at this stage.
- **Look At:** Determines the look at behavior of the camera at this stage.
- **Z Tilt:** Determines the euler Z angle of the camera at this stage.
- **Shake:** Determines the camera shake effect at this stage.

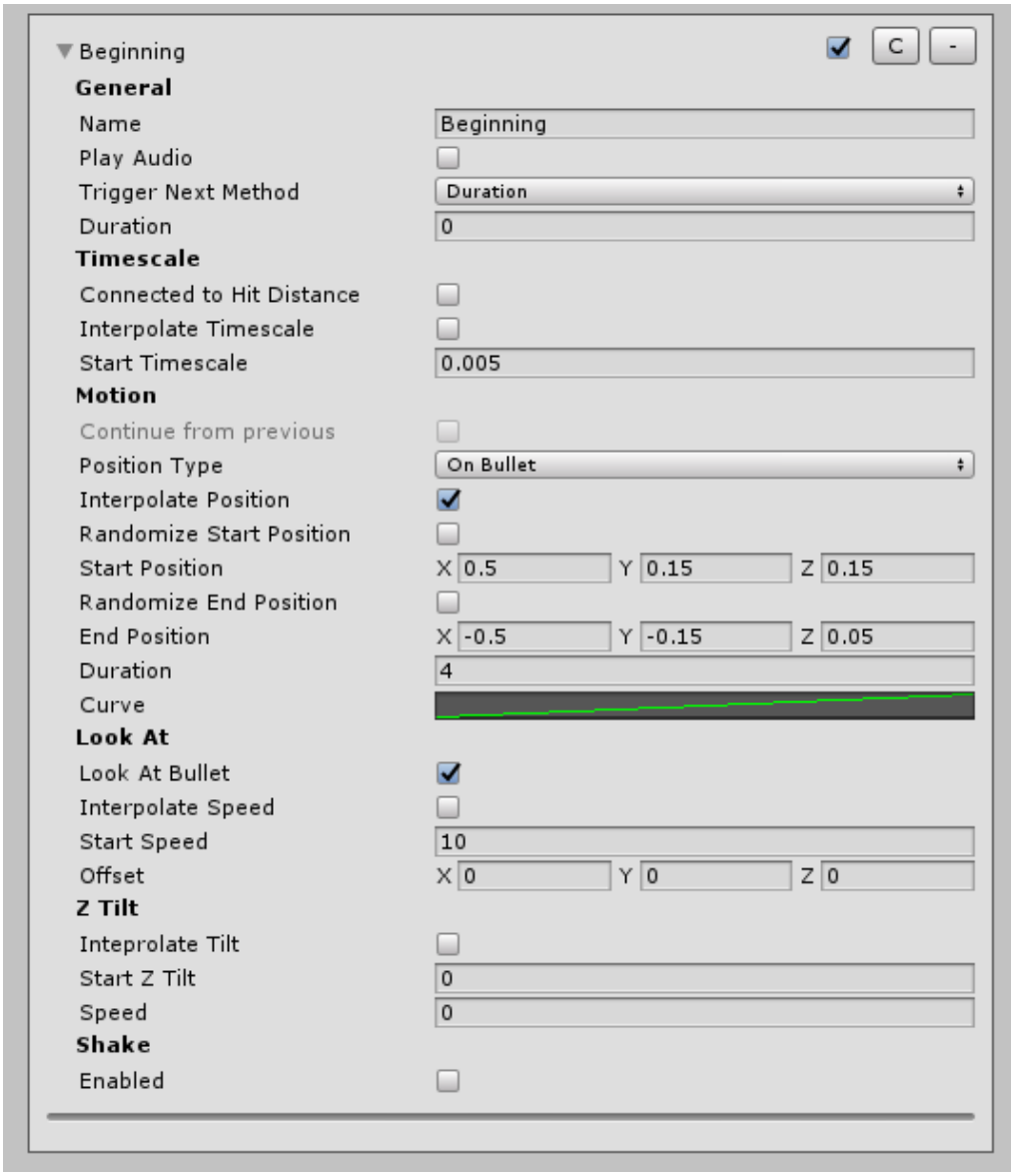
So now let's head over to the **Motion** section and set the start position as 0.5, 0, 0. Then hit play, and we will see this when we fire:



Our bullet time effect being triggered when the cube with the BulletTimeTarget.cs is hit.

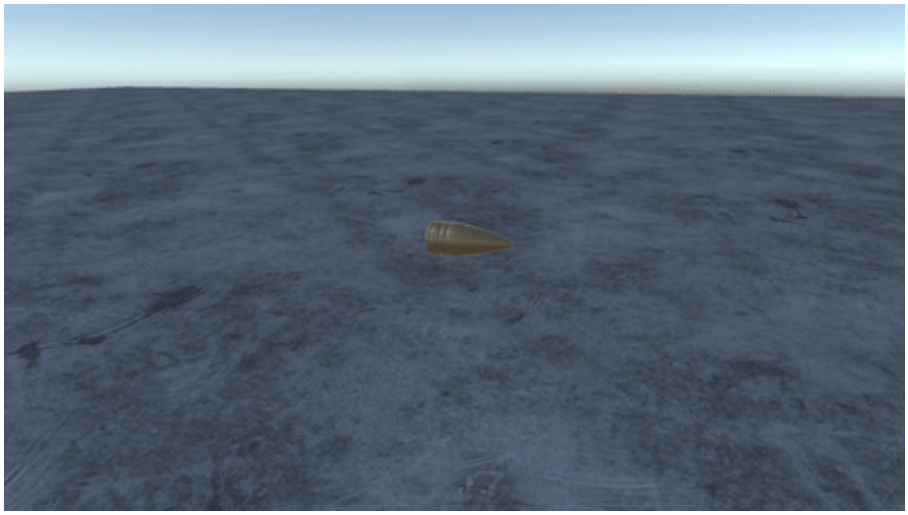
However, you will realize it will get stuck like this. This is because the **Timescale** of our stage is set to 0. Change the **Start Timescale** to **0.05** and play & fire again. You will see that the bullet will travel, camera will watch from the side and the main camera will be enabled back after bullet hits the target. This doesn't look good though, so let's customize this stage.

First, let's change the floor material so that we can clearly see that the bullet is moving. Click on the floor plane, then click on the material search on it's Mesh Renderer, select the material called **Concrete**. Next, go back to our Bullet Time Camera object, and continue editing our stage. Let's set the **Start Timescale** to **0.005**. Then, enable **Interpolate Position** checkmark in **Motion** section, set the **Start Position** to 0.5, 0.15, 0.15 then set the **End Position** to -0.5, -0.15, 0.05. Then set the **Duration** to 4, click on the **Curve** and set a linear curve. This basically offsets the camera from the bullet by Start Position, then interpolates the camera towards the offset defined by End Position using the duration and the curve. So the whole stage should look like this:



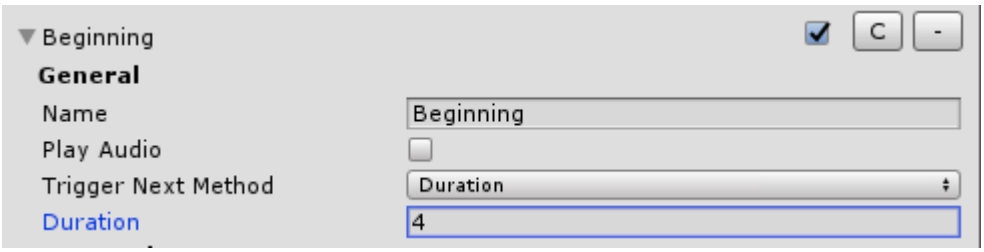
Position interpolated, start timescale changed.

If you play and shoot now, you will see that our bullet time camera moves from one side of the bullet to the other.



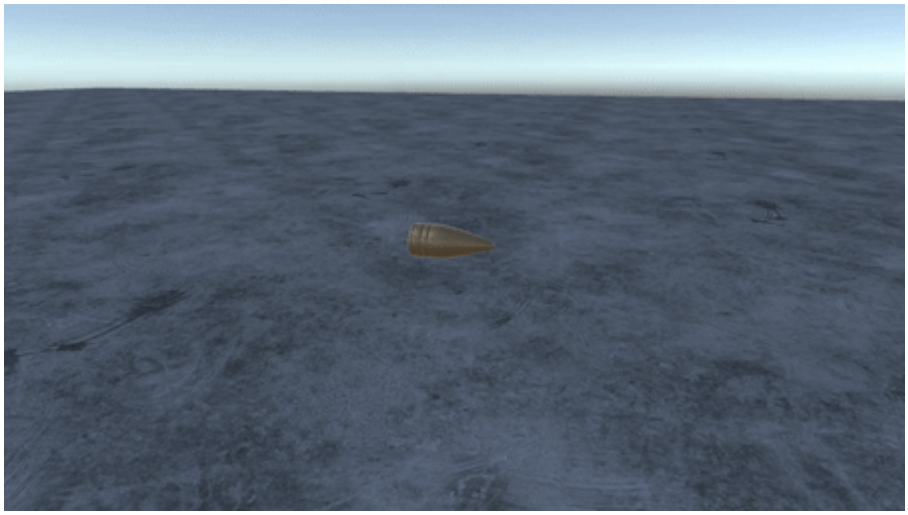
Bullet time camera moving from one side to the other.

Now let's focus on creating a stage and switching stages. The idea is that we made a stage for the beginning, with a relatively low timescale. Now we want to have a middle stage, where the bullet travels with a higher timescale and we follow the bullet from behind. However, first we need to define how to trigger the next stage. There are multiple methods to do this, but we will use a fixed duration for this tutorial. Make sure that the **Trigger Next Method** field is set to **Duration** in **General** section, and the **Duration** value is set to **4.**



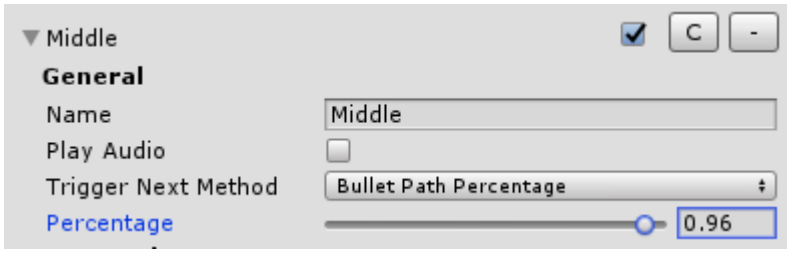
Next stage will be triggered 4 seconds after this stage is initiated.

Next, close the **Beginning** stage foldout, click on **Create New** button on **Stages** field to create a new stage, and rename it to **Middle.** Set the **Start Timescale** to **0.09**. Check the **Continue From Previous** checkmark in **Motion** stage. This will ensure when this stage is triggered, the camera position starts from wherever the camera was at the end of the last stage. Then check the **Interpolate Position** checkmark, and set the **End Position** to **0, 0.1, -0.2**. Set the interpolation duration to **1**, and again, set a linear curve. If you play and shoot now, you will see the camera will travel to the back of the bullet when this stage is triggered.



Bullet time camera moving backwards when second stage is triggered.

Now as you will see the bullet travels relatively fast when the second stage is triggered, but it still looks bad while it's about to hit the target. So, head over to the **General** section in **Middle** stage, change the **Trigger Next** method to **Bullet Path Percentage** and set the **Percentage** value to **0.96.**

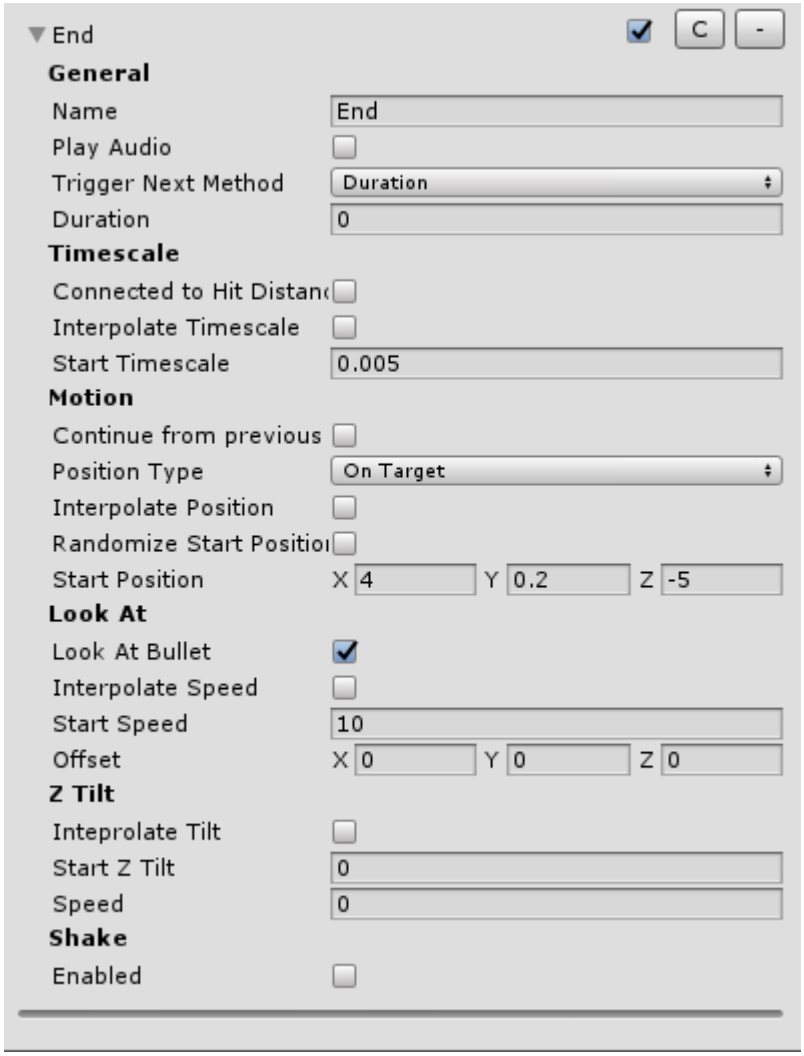


Trigger next method is set to path percetange.

This will make the system trigger to the next stage when the bullet has traveled 96% of it's path. Then, close the foldout, create a new stage and name it **End.** Set the **Start Timescale** to **0.005,** go to the **Motion** section and make sure **Continue From Previous** and **Interpolate Position** are **unchecked** this time. Set the **Start Position** to **4, 0.5, -5**. Set the **Position Type** to **On Target**.

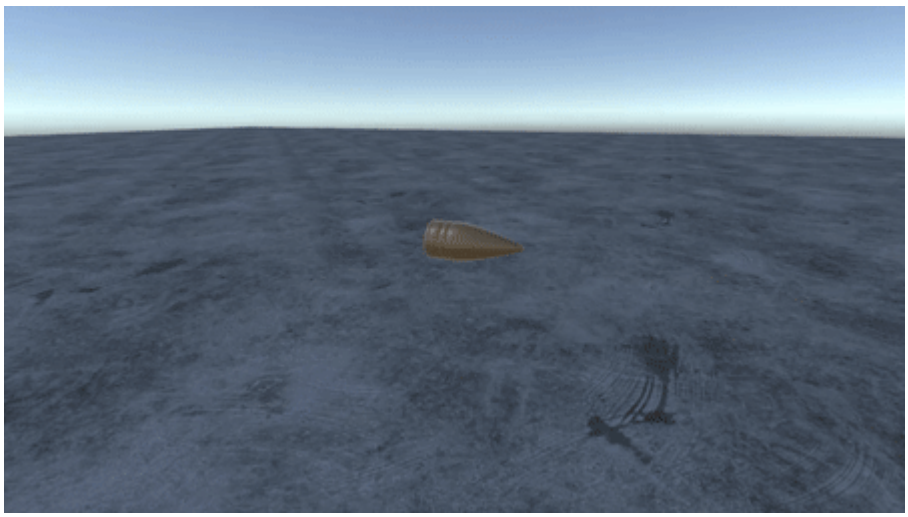
> The position values work as **offsets** from the **target**. By changing the **Position Type** to **OnTarget**, we ask the system to put the camera at the center of the **hit target**, then offset it by the position amounts.

So the **End** stage should look like this:



> As this will be the final stage, we can omit the **Trigger Next Method** in this case. Since there will not be a next stage, the system will not try to trigger anything.

If we play and shoot now, this is what we will see:



Bullet switching to final stage when it's about to hit the target, then watching the bullet from the side.

To summarize the logic, we can list:

- Bullet time event gets triggered, our **BulletTimeCameraDefault.cs** picks this event, then looks for an effect to play.
- We created a new effect called **Tutorial Effect** and set this to be the **Used Effect**. Now the bullet time camera will start this effect when an object with **BulletTimeTarget.cs** is hit.
- First, the **Beginning** stage is triggered, travelling from one side of the bullet to the other.
- After 4 seconds has passed, **Middle** stage is triggered, switching to the back of the bullet.
- After the bullet has travelled 96% of it's path, **End** stage is triggered, watching the bullet by placing itself next to the target.

By leveraging the full features of this tool, it is possible to create extremely powerful bullet time effects. All other sections work the same way, most values are set to the start values at the beginning, and they can be interpolated via a specific duration and an animation curve. Play with other sections, such as **Camera Shake** or **Z Tilt** to get an idea of what this tool of capable of.

RSB already provides you with a bullet time camera default prefab that includes **10** different bullet time effects. This prefab can be found under the folder **InanEvin>Realistic Sniper and Ballistics>System Prefabs** with the name **Bullet Time - Camera Default Presets.** This prefab is also used in demo scenes. It is suggested to keep a copy of this

prefab somewhere safe, so that if you make any changes to it you won't lose the original presets that come with the package.

Writing your Own Bullet Time Camera

The **Bullet Time Camera Default** is an intuitive and powerful tool to create bullet time effects that will fit to most gameplay scenarios. However, if you want to write your own camera effect, code the logic yourself, **RSB** provides the means to do so. There are 4 events in **SniperAndBallisticsSystem.cs** that will be fired during a bullet time:

- **EBulletTimeStarted:** Fired when a **BulletTimeTarget.cs** object is hit, bullet object is enabled, ready to be travelled through it's path.
- **EBulletTimeUpdated:** Fired when the bullet travels through it's path.
- **EBulletTimePathFinished:** Fired when bullet hit's the target.
- **EBulletTimeEnded:** Fired when the timescales are set back to normal and everything is resetted.

By creating a script that listens to this events, you can enable a camera when the bullet time is triggered, move the camera during the update events, and disable the camera when it ends. You can fill in the logic in anyway you want in between. There exists a template class to do this, navigate to **InanEvin>Realistic Sniper and Ballistics>src>BulletTime** folder and take a look at **BulletTimeCameraEmpty.cs** file. This provides the baseline support for writing a simple bullet time camera. It's a good starting point, so if you want to try writing your own logic, you should start by copying this template.

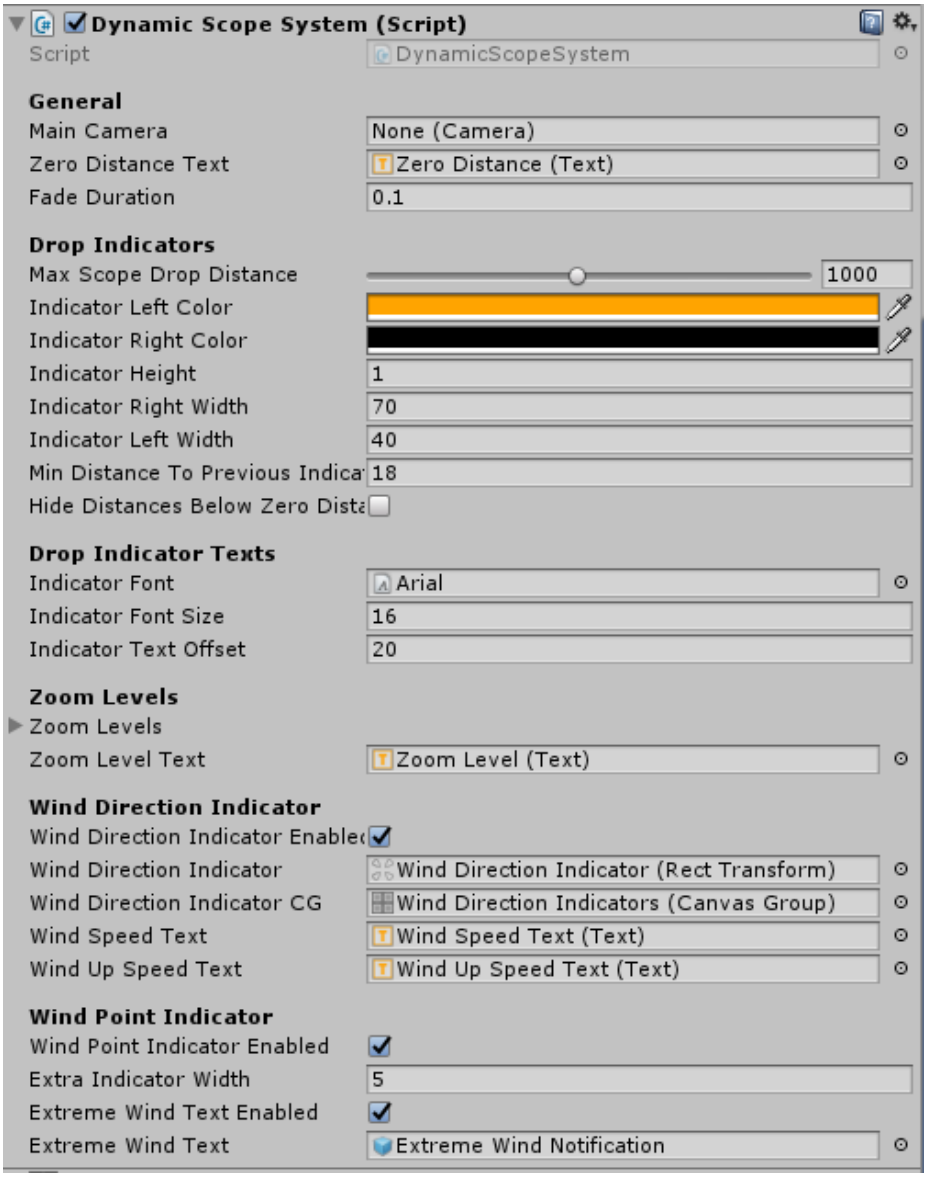
Dynamic Scope System

So far, we have learned how to fire a bullet using SniperAndBallisticsSystem.cs instance, listen to the hit events, how to create ricochet-able and penetrable objects using Ballistic Surfaces, and how to incorporate Bullet Time Effects into our scene. Now let's take a look at another powerful tool provided by RSB, **Dynamic Scope System.**

Dynamic Scope System (**DSS)** is a Unity Canvas based tool that displays a sniper scope texture, along with how much will the bullet drop, dynamically based on current zoom levels and zero distances. It also has a wind support, displaying where the bullet will land with the current wind settings.

Before incorporating DSS into our scene, let's change it a bit. We had a cube object positioned at 0,0,100 and it had a BulletTimeTarget.cs on it. Let's rename this cube as **Bullet Time Cube,** then move it to the position **15,0,100.** So that it's out of our camera's way. Then let's create a new cube, call it **Default Target**, then place it to **0,0,200.** Also scale this new cube to **3,5,3.** Now, we are ready to use DSS.

We can create our own canvas setup, attach the **DynamicScopeSystem.cs** component on it & setup the necessary variables. However, RSB already provides a completely ready-to-use prefab for this purpose. Navigate to **InanEvin>Realistic Sniper and Ballistics>System Prefabs** folder and drag & drop the **CANVAS - Dynamic Scope System** prefab into the scene. Open up the canvas foldout, and select the **ScopeReticle** object. There you will see the **DynamicScopeSystem.cs** component.



DynamicScopeSystem.cs component on ScopeReticle object under CANVAS - Dynamic Scope System

Head over to the **General** section, and attach our main camera to **Main Camera** field. If you play the scene now, there will not be any changes.

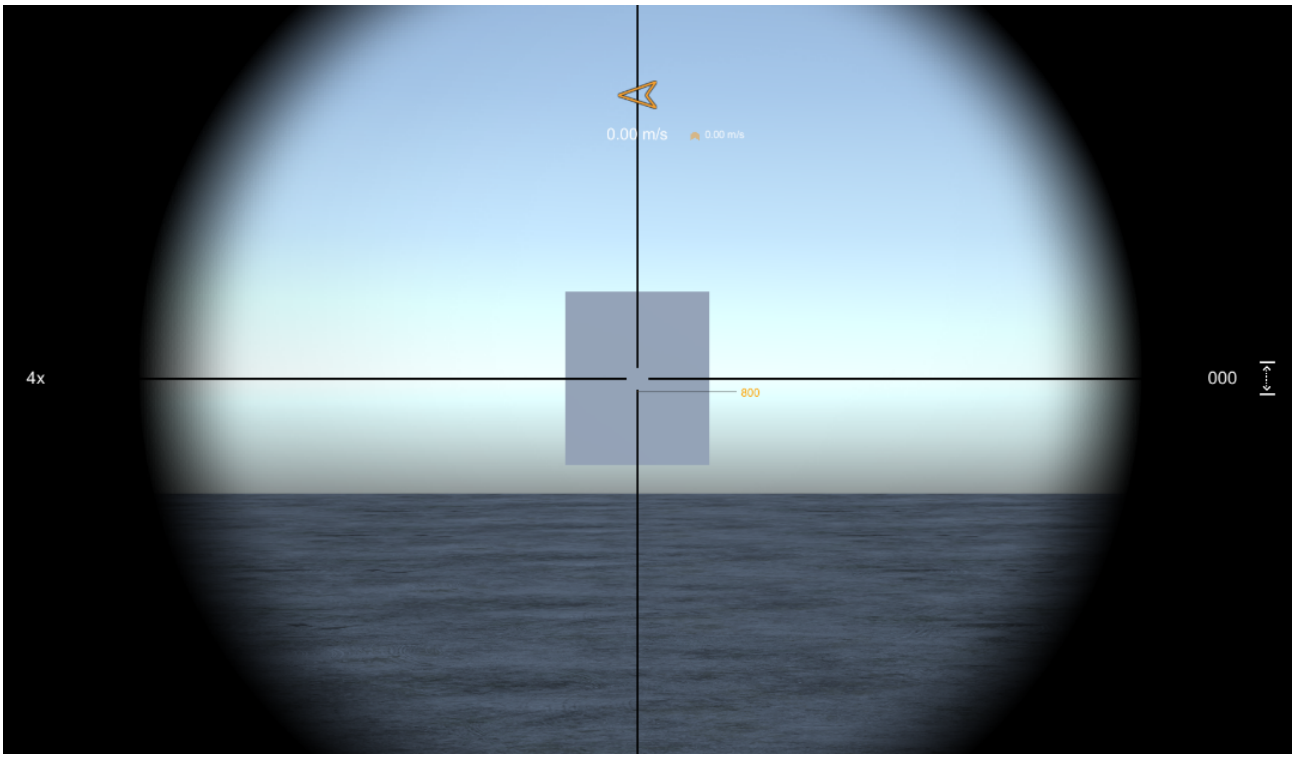
DSS has a public method we can call to enable the scope reticle. Open up our **Tutorial.cs** script and let's write two input checks in our Update to enable & disable the scope.

RSB-Tutorial16.cs https://gist.github.com/3fbfce719e4f2f4f125b6fb91360d9ed.git

So, we want to enable the scope when we right-click and hold, then we want to disable it when we release right-click. We don't need a reference to the DSS as it is a singleton object, just like SniperAndBallisticsSystem. We can call the **ScopeActivation** method in DSS to activate/deactivate the scope reticle.

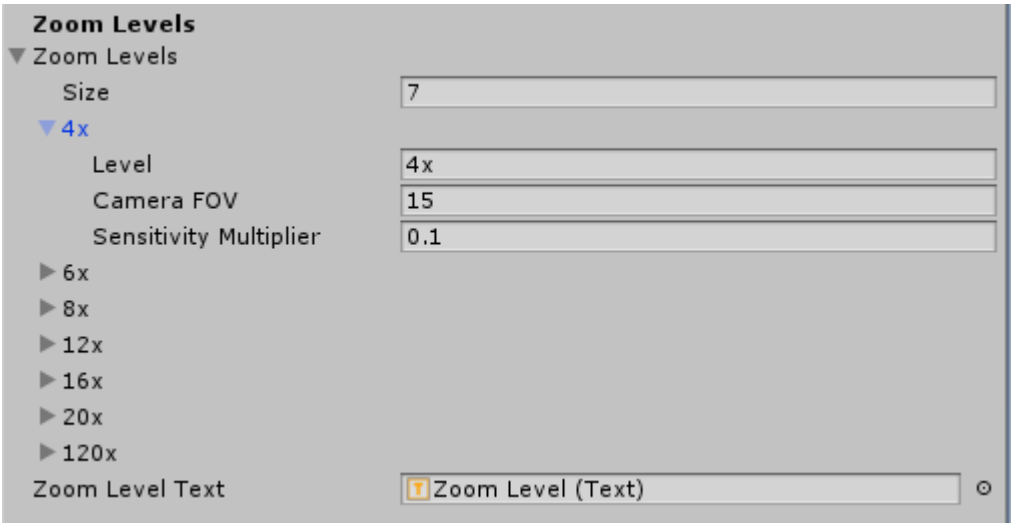
RSB-Tutorial17.cs https://gist.github.com/dedc6b8f044f7f45d2421d96dfc36f34.git

ScopeActivation method receives a boolean to know whether to activate/deactivate the scope. It also requires a BulletProperties reference, so that it can setup the bullet drop distances & trajectory specific information accordingly. If we play the game now, and click&hold right-click, we will see the scope activated.



DSS activated scope.

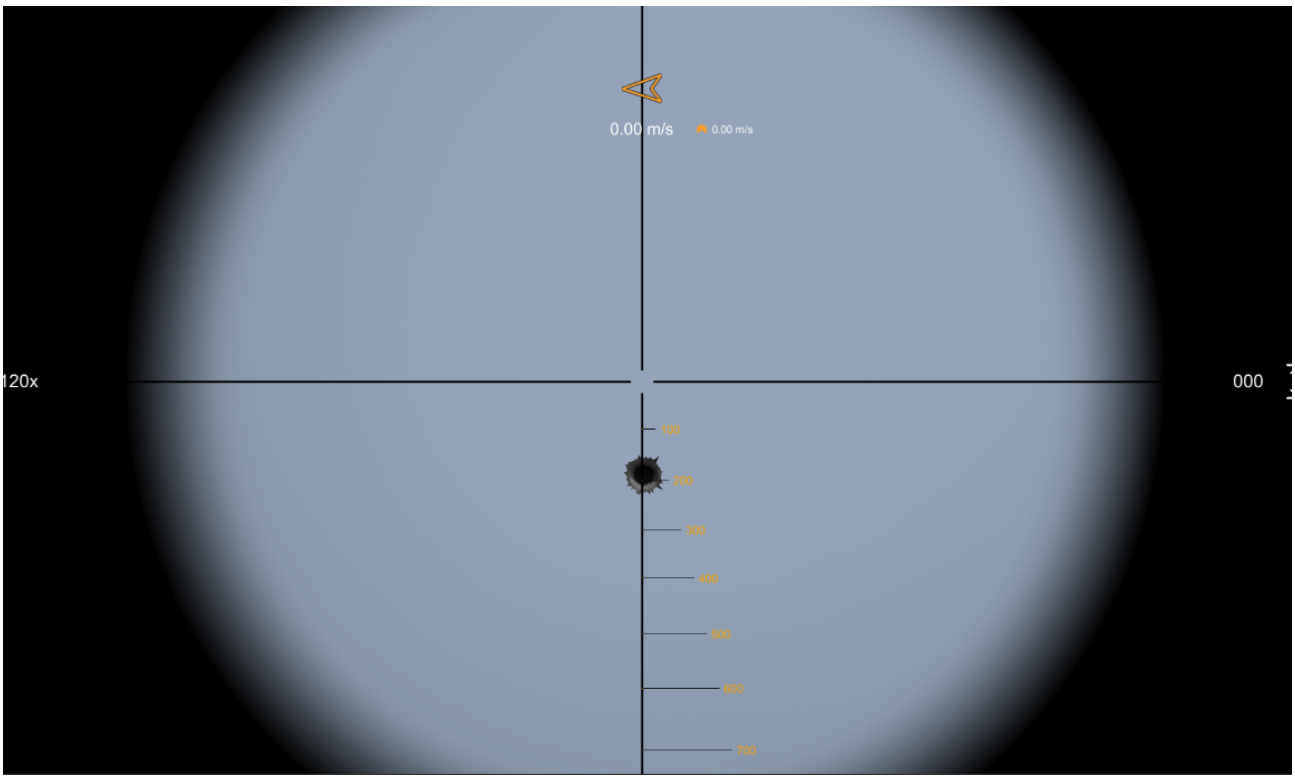
By using **mouse wheel,** you can change zoom levels. Each zoom level is set over the **DynamicScopeSystem.cs** component.



Different zoom levels in DynamicScopeSystem.cs component.

Each zoom level has a display name, camera's target field of view, as well as sensitivity multiplier. The static float variable **DynamicScopeSystem.ScopeSensitivityMultiplier** will be set according to the multiplier of the current zoom level. You can use this static variable to change your aim sensitivity during active scope. An example of this is used in **PlayerCameraController.cs** script, under the folder **InanEvin>Realistic Sniper and Ballistics>src>Player**.

When you zoom in/our while the scope is active, you will realize that the drop indicator lines at the bottom change. Each line has a distance text next to it. They determine where the bullet will hit (how it will drop) at that specific distance. For instance, if we zoom in really close to our target, hit space bar, you will see that the bullet doesn't hit to the middle of the crosshair, but it hits where the **200** meter drop indicator marks.



Bullet hitting 200 meter mark.

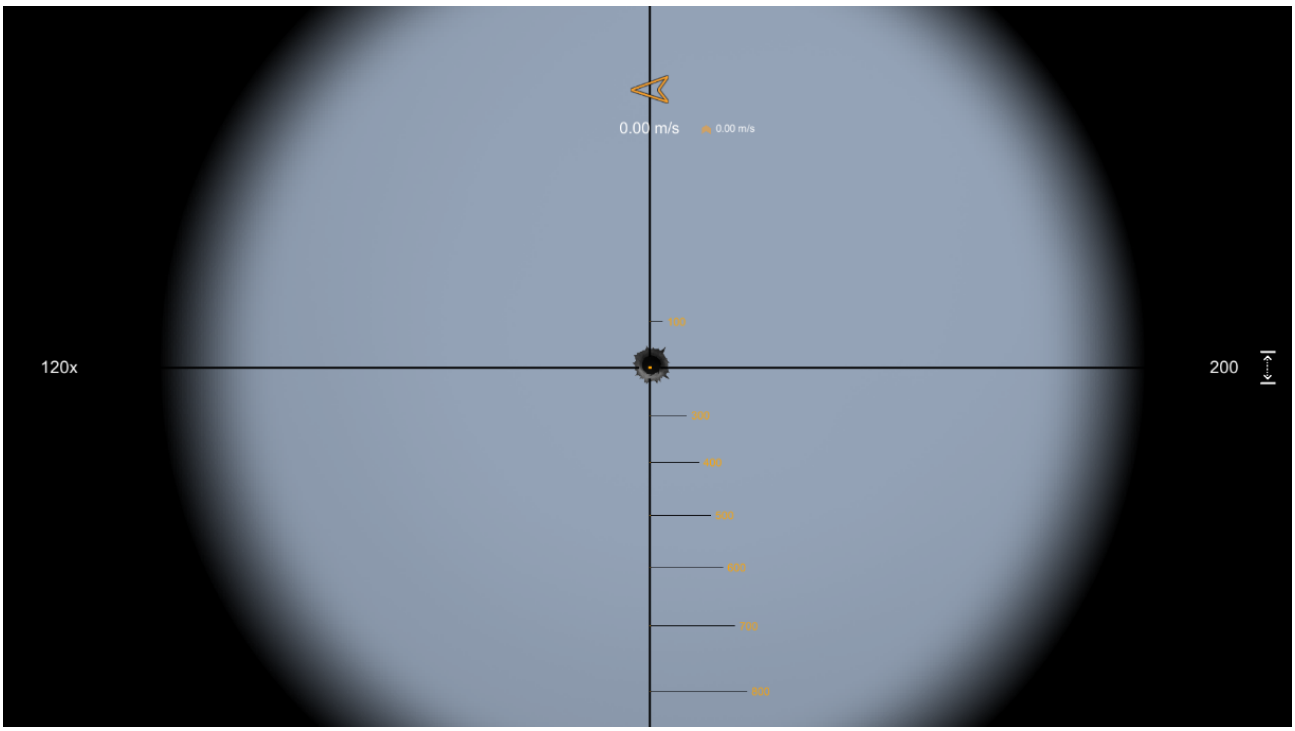
This is natural, as the bullet will drop along it's path. DSS is a powerful tool that can show these drops in a 2D manner. It also supports **zeroing**, so if we change our zero distance, it will dynamically scale the drop distances. Remember we had already written this in **Zero Distances** section in this tutorial series.

RSB-Tutorial18.cs	https://gist.github.com/abd9a1bea90b590fb50930bf96239328.git

So our whole **Tutorial.cs** script should look like this:

RSB-Tutorial19.cs	https://gist.github.com/abefe1f7bca5443c7713c602d8505387.git

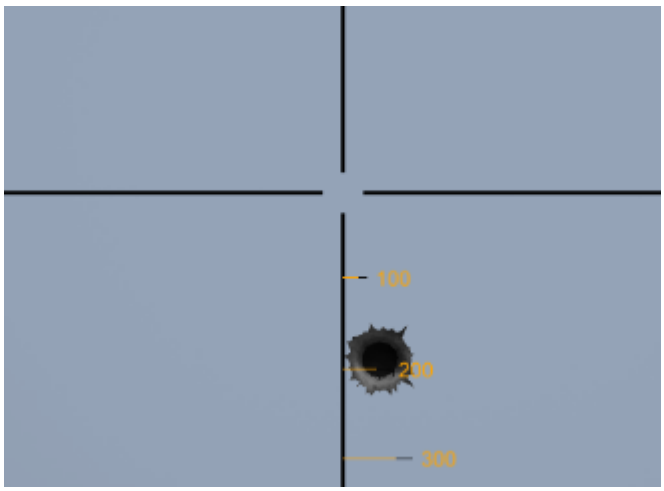
Now if we play our game, zoom in close to the target, then press **C** key to cycle the zero distance up, you will see the zero distance changing on the right side of the scope. Set the zero to **200,** then fire.



Firing at a target at 200 meters, with a zero distance set to 200 meters.

As you can see, now we are hitting exactly where we are aiming at. If we try to hit a target further away from 200 meters, the bullet will still drop, which can be visualized again by the drop indicators below. If we try to hit a target closer than 200 meters, while the zero is set to 200 meters, the bullet will go **above** where we are aiming at, which is again visualized by the drop indicators.

One other powerful feature of DSS is that it can show the wind effect on the bullet. Right now, our Environment Properties asset does not have any wind speed, so let's change that. Find the Environment Properties asset you have created for this tutorial (you can select the Sniper And Ballistics object in the scene, then find the asset from there since it was referenced). Change the wind speed to **90, 0,0.** Now play the game, zoom in, then fire, you will see the bullet will hit to the right side of the target's center. And the **orange lines** in drop indicators will indicate where the bullet will land according to the wind.



Orange lines indicating bullet's landing position with the drop distances.

If your wind vector is too powerful, the bullet might land somewhere invisible by the camera (way too far off to the right/left, depending on the wind vector). In those cases, it would be impossible to show where the bullet will land in Screen Space, thus in those scenarios, DSS clamps the position of the orange lines, and displays a "Extreme Wind" text on the screen. This is less of a gameplay feature, but more of a developer tip indicating to you that maybe you have exaggerated the wind vector.

Of course all of this functionality in DSS is completely tweakable over the **DynamicScopeSystem.cs** component. You can change the line colors, line thickness and maximum widths, along with you can determine what is the maximum drop distance that will be shown, how much gap there will be between the drop indicators, whether to show the wind indicator on top, or whether to show the wind point indicators (orange lines) in the screen. Play with it just to see what can be achieved.

It is suggested to keep a copy of the original prefab in somewhere safe, so your modifications won't change the original DSS preset that comes with the package.

Incorporating Player

So far, the information we have learned should be enough for you to easily start using RSB in your own player controllers & character systems. However, if you don't have your own character system, or if you want to build upon a base, you can use the example first-person controller provided by RSB. Let's change the tutorial scene such that instead of firing from the main camera, we will fire from the player camera.

First, delete the Main Camera object. Then navigate to **InanEvin>Realistic Sniper and Ballistics>System Prefabs** folder and drag & drop the **Player Without Weapon** prefab into the scene. Position it to **0,1,0.** Then go over our **Tutorial** object, you will see that the Main Camera is missing. Drag & drop the **Main Camera** under the player prefab in the hierarchy. Then go to the **Sniper and Ballistics** object and assign the same **Main Camera** to the **Fire Transform** field of the SniperAndBallisticsSystem.cs instance. Next, go over to the **ScopeReticle** object under **CANVAS - Dynamic Scope System** where the DynamicScopeSystem.cs component is attached, and assign the same **Main Camera** under the player to the **Main Camera** field. Finally, open up our **Tutorial.cs** script, and modify the if block where we are firing to:

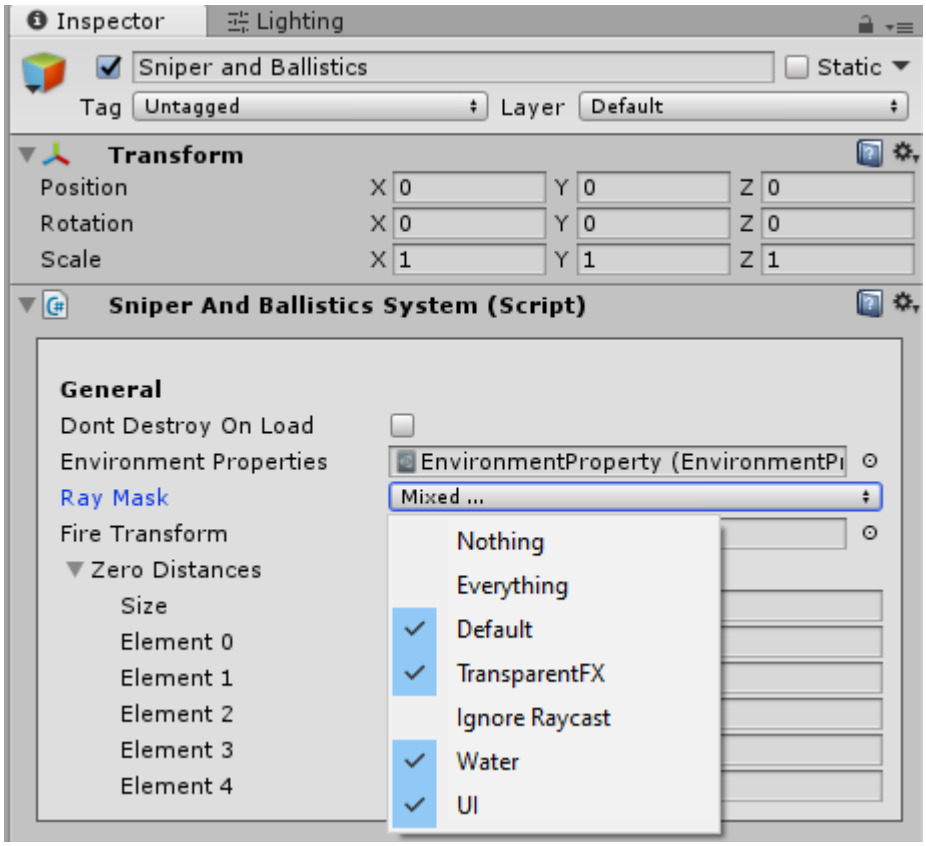
RSB-Tutorial23cs

https://gist.github.com/261333f5ae24fda45a675c455f6ea6e8.git

Remember we used to fire using **Space** key, but that will conflict with the **Jump** input of the player controller we just dragged in. Now, we have just switched that to the left-click mouse button.

There is only a small thing left to do. The **Player without Weapon** prefab, as well as the other player prefab we will talk about in the upcoming sessions, are **layered** as **IgnoreRaycast**. This is only for the sake of the demo scenes, of course in your own project, you might want to raycast to the player, e.g. from an enemy, so you might want to put the player into some new layer like **Player**. But for the RSB package, we did not want to mess with your project settings & layers, so we have just put the player under **IgnoreRaycast.** Now, it is possible that when we fire from the player, the rays might hit any object under the player, if we were to have an object with a collider. That's why we want to bypass the player

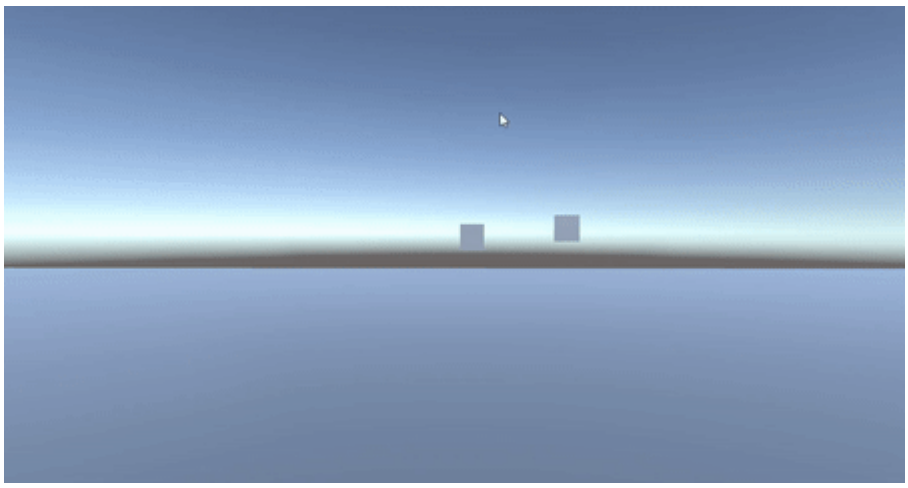
during our raycasts. Go to the **Sniper and Ballistics** object, and change the **Ray Mask** value to **Everything but IgnoreRaycast (Mixed).**



Every layer except IgnoreRaycast is selected.

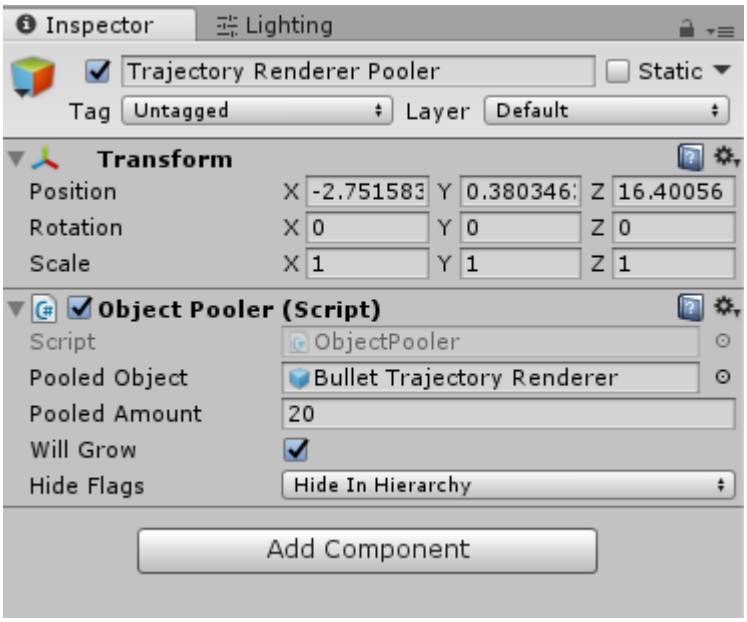
Since the player is in IgnoreRaycast layer, now the rays casted from our fire transform has no chance of hitting the player collider, or any object we put under the player if there were to be one.

If you play your scene now, you will be able to walk around, and shoot with the mouse left-click. Remember, you can also aim with right-click.



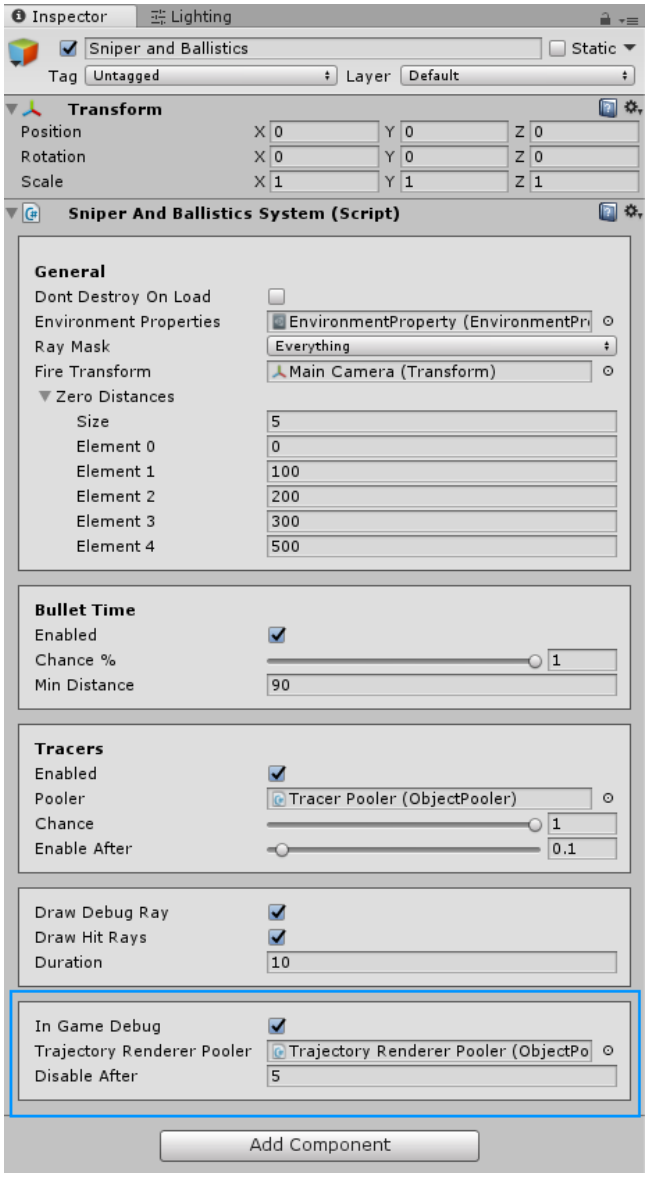
Shooting with the example player controller.

Let's also debug the bullet trajectory in-game. Create a new game object, call it **Trajectory Renderer Pooler,** attach the **ObjectPooler.cs** component on it. Then click on the search field of the **Pooled Object** field and assign the **Bullet Trajectory Renderer** prefab on it.



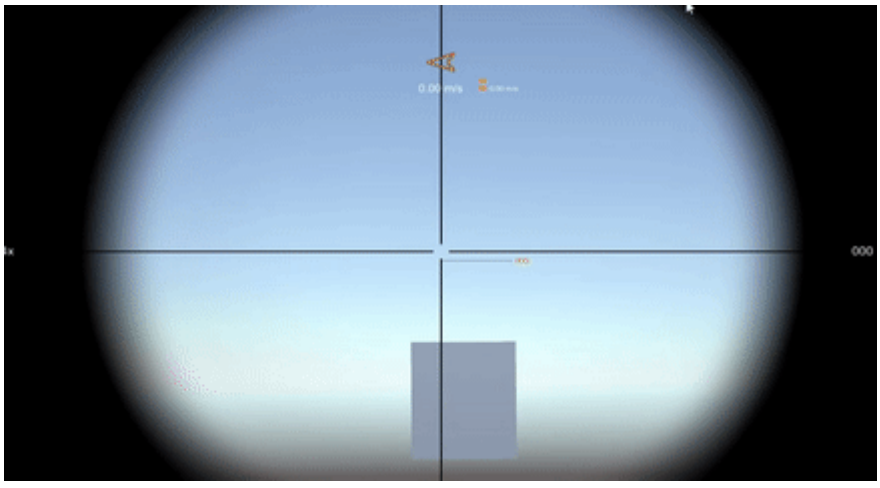
New object created & ObjectPooler.cs attached, with Bullet Trajectory Renderer set as pooled prefab.

Then go over to the Sniper and Ballistics object, enable **In-game Debug** renderer settings, and assign this pooler to the **Trajectory Renderer Pooler** field.



In-game Debug enabled, trajectory renderer pooler assigned.

Now if you play the game and shoot, you will see the bullet's trajectory being rendered with line renderers.

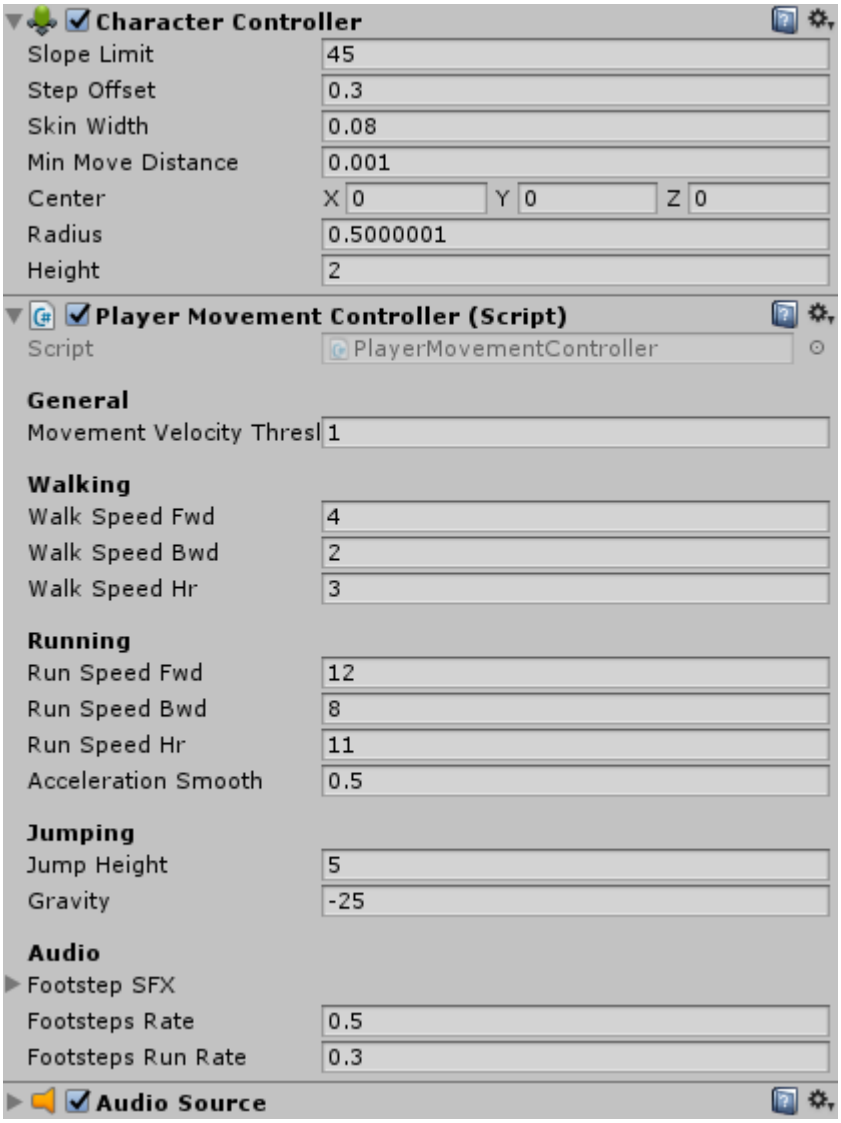


In-game trajectory rendering.

Let's take a look at the details of the player prefab we have just dragged in, just to understand how it works.

Player Movement Controller

If you click on the **Player Without Weapon** prefab we have put into our scene, you will see that it has a **Character Controller** and a **PlayerMovementController.cs** component. This is what makes our player walk, sprint & jump.

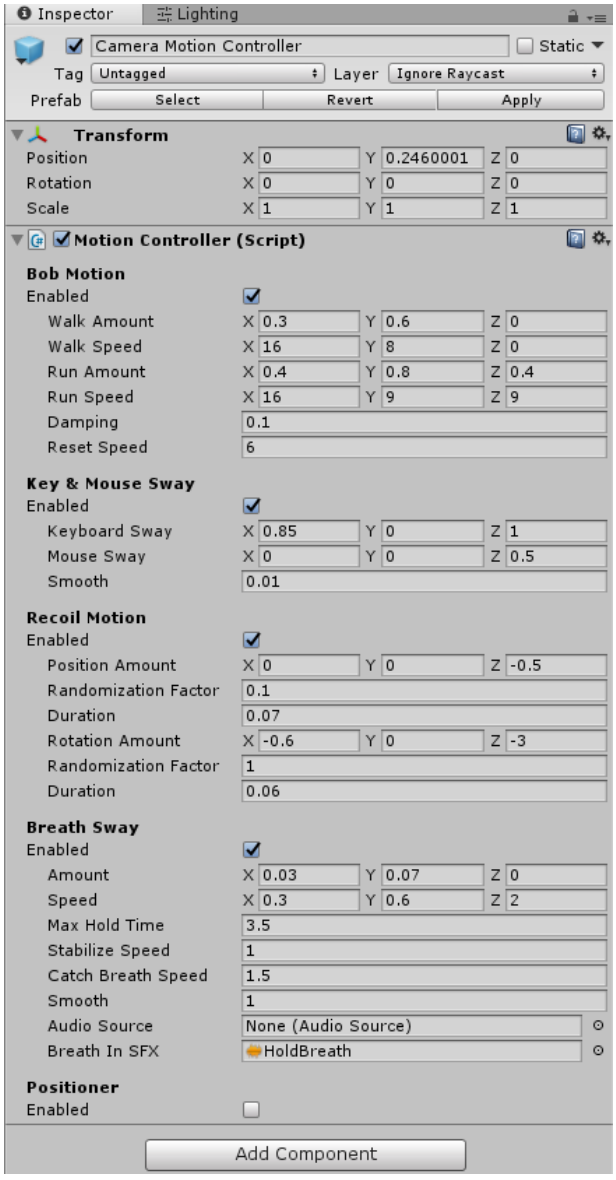


PlayerMovementController.cs

You can take a look at the inside of **PlayerMovementController.cs** script to see how the movement is handled. It has a very basic state-machine setup, so that we can listen to events to see when the player's movement state has changed. You can play with the movement speeds, jump height, gravity etc. to change the behavior of the movement script.

Motion Controller

If you continue looking down the hirarchy of the **Player Without Weapon** prefab, you will see a **Camera Animator** object. This is simply an example animator that animates a "camera-kick" effect when the player fires. Below that, we have **Camera Motion Controller** object, which has **MotionController.cs** component attached to it.



MotionController.cs attached to the child object of Camera Animator, which is a child of Player Without Weapon prefab.

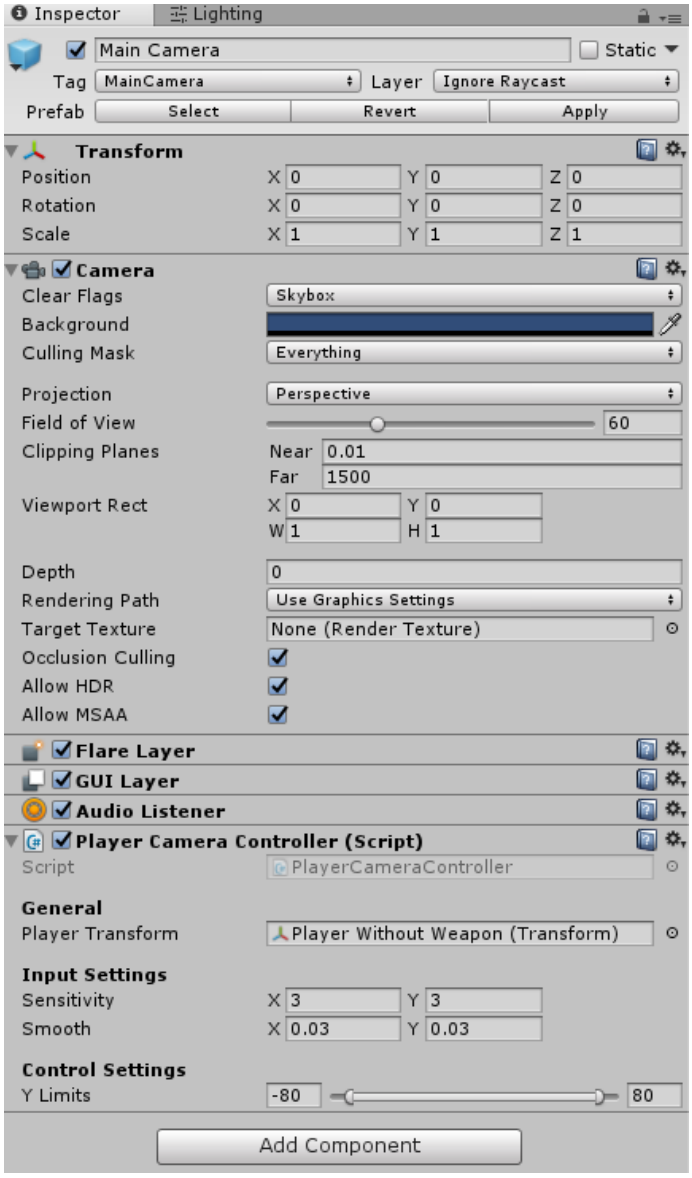
This motion controller is the script that handles various effects to give life to the camera object.

- **BobMotion:** Bobs the camera with sine waves based on whether the player is walking or running.
- **Key & Mouse Sway:** Sways the camera's rotation based on keyboard & mouse input.
- **Recoil Motion:** Recoils camera when player fires.
- **Breath Sway:** When player aims, enables a sway effect mimicking player moving the weapon due to breathing. Player can hold **shift** to stabilize.
- **Positioner:** Can place the object to **hip, aim** or **running position,** based on whatever method is called. Is not being used in here, but will be used for weapon.

BobMotion and **Key&Mouse Sway** works individually, one of them is based on player movement events coming from **PlayerMovementController.cs** and the other one is solely based on key&mouse inputs. **Recoil Motion** requires you to call **Recoil()** method in **MotionController.cs**, same for using **BreathSway** and **Positioner** behaviors. Since our player prefab doesn't have a weapon now, these effects are enabled but there is no-one calling them.

Camera Controller

Under the **Camera Motion Controller** object, we have our **Main Camera**, which has a **PlayerCameraController.cs** script attached to it. This script is a smooth camera controller for first-person games.



PlayerCameraController.cs in Main Camera.

For now, our player does have the shooting functionality, thanks to our **Tutorial.cs** script, but it doesn't have a weapon. Let's change that in the next section.

Player With Weapon

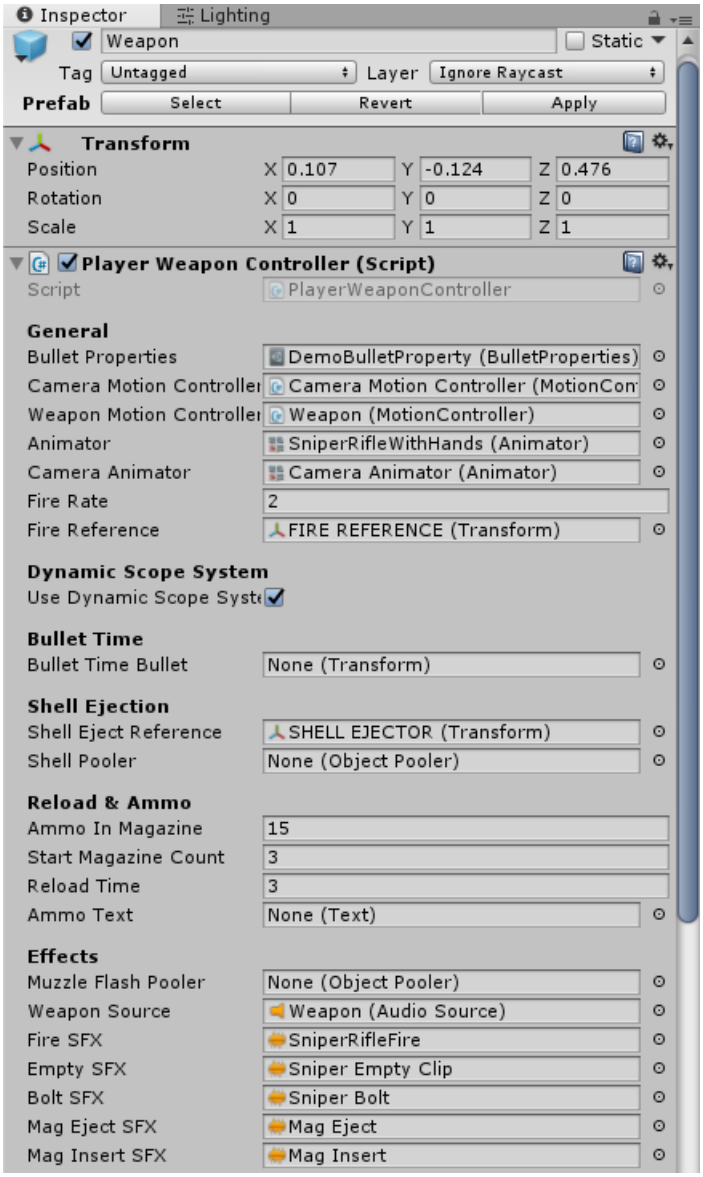
Since we understood the basics of how to incorporate a player object with RSB, now we can use our prefab that actually has a weapon controller. **Delete** the **Player Without Weapon** object from hierarchy, then navigate to **InanEvin>Realistic Sniper and Ballistics>System Prefabs** folder & drag & drop the **Player With Weapon** prefab into the scene. Set it's position to **0,1,0.** At this point, we do not need our **Tutorial.cs** script to fire, aim, or change the zero distance for us, as this will be done by the **PlayerWeaponController.cs** coming with the **Player With Weapon** prefab. So, let's open up our **Tutorial.cs** and comment out the unnecessary lines:

RSB-Tutorial24.cs https://gist.github.com/c59f1672b21c5405c41c7bf9aa7afb5f.git

Now our **Tutorial.cs** script is only responsible for spawning hit effects.

Just as the beginning of the previous section, we'd have lost references to the **Main Camera**. So, go to **Sniper and Ballistics** object, assing the **Main Camera** under the new **Player With Weapon** object in the scene to the **Fire Transform** field of SniperAndBallisticsSystem.cs instance. Then, go to the **ScopeReticle** object and assign the same **Main Camera** to the **Main Camera** field of **DynamicScopeSystem.cs** instance.

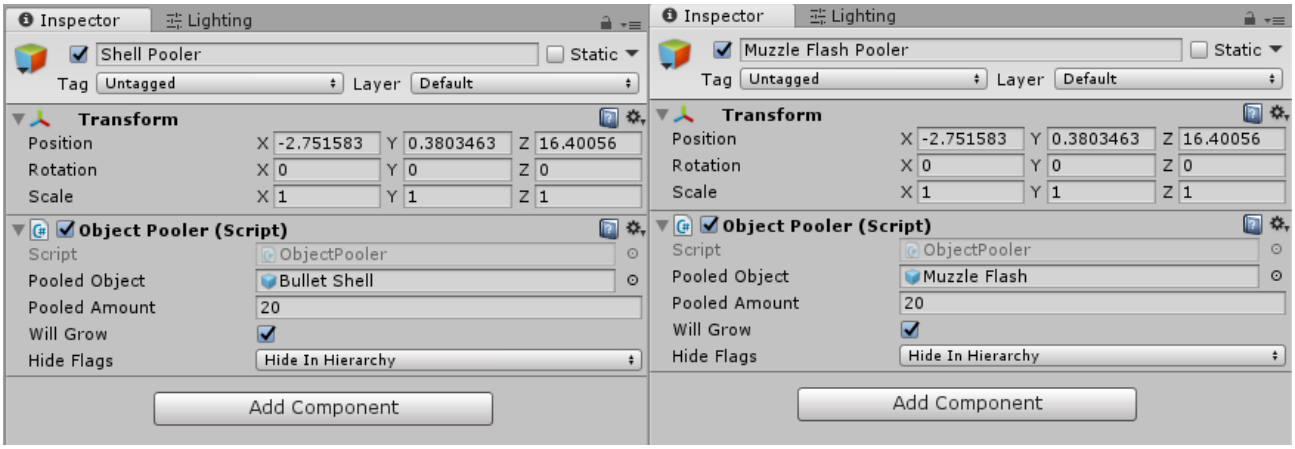
Now if you open up the hierarchy of **Player With Weapon** object, you will see under the Main Camera, there is a **Weapon** object, that has a **PlayerWeaponController.cs** component attached to it, as well as MotionController.cs attached to it in order to control the weapon's movement.



PlayerWeaponController.cs attached to Weapon object.

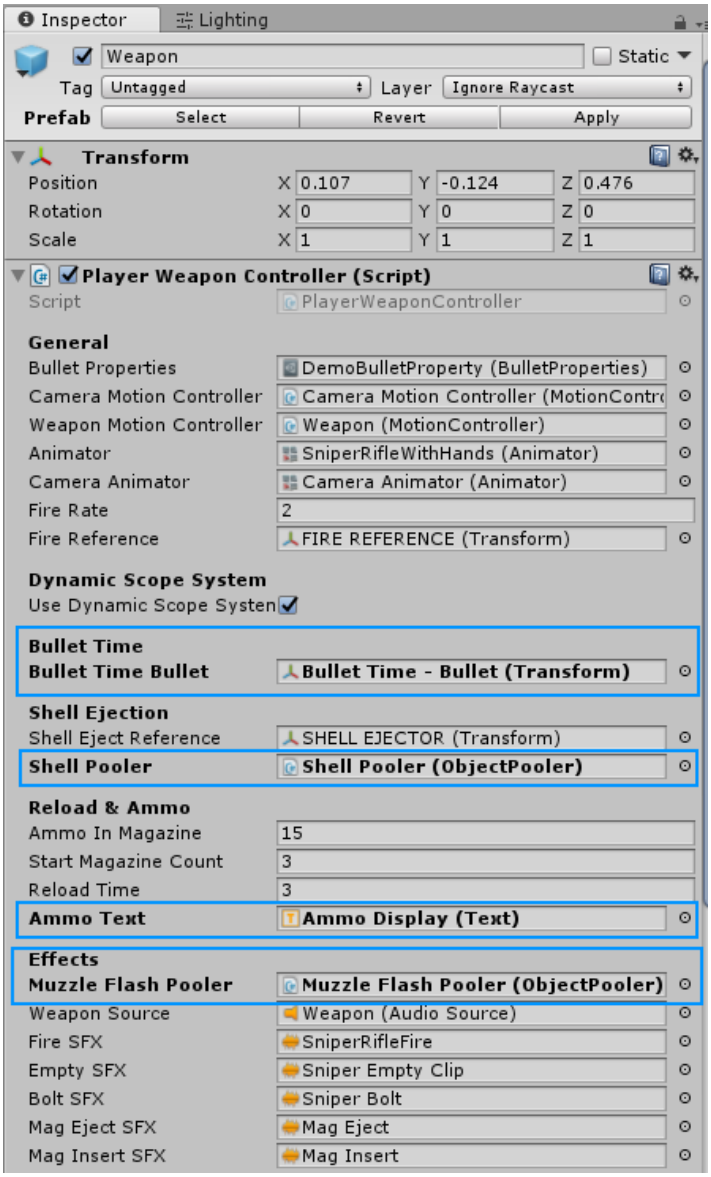
PlayerWeaponController.cs does exactly what we did in our **Tutorial.cs**, just in a more complicated fashion in order to create the behavior of a bolt-action sniper rifle. Before we can try this out, there are couple of fields we need to fill. First, drag & drop the **Bullet Time - Bullet** object in the scene to the **Bullet Time Bullet** field of **PlayerWeaponController.cs** component.

Then, we need to create 2 new poolers, so create 2 new empty game objects, rename them as **Shell Pooler** and **Muzzle Flash Pooler**, then add the **ObjectPooler.cs** component on them. At this point, you should have been pretty comfortable adding new poolers, so go ahead and assign the necessary prefabs for them. **Shell Pooler** will use the prefab **Bullet Shell**, and the **Muzzle Flash Pooler** will use the prefab **Muzzle Flash.**



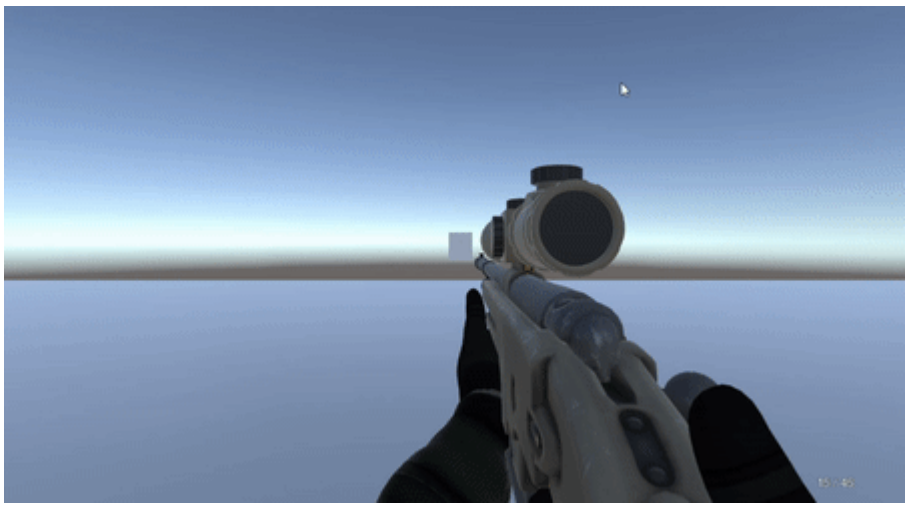
Muzzle Flash & Shell poolers.

Go back to our **PlayerWeaponController.cs** component and attach these poolers to their respective fields. Then lastly, we need a **text** component in the scene to display the remaining ammo. Navigate to **InanEvin>Realistic Sniper and Ballistics>System Prefabs** folder and drag & drop the prefab called **CANVAS - Crosshair & Ammo** into the scene. This is just a very simple Unity Canvas that has an image for a crosshair and a text for ammo display. Drag & drop the text called **Ammo Display** under this canvas to the **Ammo Text** field in **PlayerWeaponController.cs**. So, it should look like this now:



Bullet Time Bullet, Shell Pooler, Ammo Text and Muzzle Flash Pooler fields assigned.

Now if you play the scene, you can walk, sprint & jump around, while shooting a sniper rifle. And if we hit the bullet time target, it will trigger the bullet time effect!



Mobile Controls

RSB is mobile optimized, meaning that you can use all of the features we covered in mobile.

> Only case where it might be performance heavy to use is the **Wind Point Indicators** of **Dynamic Scope System**. Remember we had those orange lines showing where the bullet will land depending on the wind? They might be a little too much for a mobile platform, however our tests using **Galaxy A70, LG G7** and **Iphone 7** showed that it worked without a problem with a stable fps.

All of the demo player & weapon controllers are suitable to use with a mobile controller. Meaning that walk input, aim input, fire input etc. are all methods that can be called from outside. If you open up the demo scene **MobileControls,** you will see that there is a canvas with a **DemoMobileController.cs** script attached, which manages a joystick input, and receives input from UI touches to aim, fire, cycle zero distances etc.

MobileDemo scene

Conclusion

So far we have learned how to setup environment & bullet properties, object poolers, use SniperAndBallisticsSystem.cs instance to fire a bullet, listen to hit events, zero distances, wind vectors, etc. as well as how to create penetration and ricochet using ballistic surfaces. Moreover, we have seen how to create our own bullet time effects and how to use Dynamic Scope System.

We have learned how we can use this information in our own player controllers & character systems, as well as how to incorporate the example player controllers provided by RSB into our scenes.

This concludes our Getting Started tutorial, you can get further information about the public API of RSB by checking the API section. Please do not hesitate to contact me through the Support Discord or my e-mail address for any further questions or feature requests.

API

API Welcome

This section will show the public methods, variables and events accessible through RSB's API. Not all the classes are included in this section, only the ones that are exposed to the users and contain important functionality.

All the classes inside the RSB package is documented within their respective files, if you can't find what you are looking for here, take a look at the script in Unity itself, you will see class description along with method explanations.

Sniper and Ballistics System

Central singleton instance for accessing ballistics functionality.

Public Variables

Type	Name	Description
float	`CurrentZeroDistance`	Current selected zero distance.
float	`BulletTimeChance`	Bullet time chance percentage between [0,1].
float	`BulletTimeVirtualTimescale`	Virtual timescale used in bullet time events. It affects the bullet's interpolation speed.
float	`BulletTimeWaitBeforeResettingTimescale`	Amount of time waited before resetting timescale when bullet time path has finished.

float	`BulletTimeWaitBeforeEnding`	Amount of time waited after timescale has been reset at the end of bullet time.
bool	`UseTracers`	Enables/disables use of tracers.
bool	`UseBulletTime`	Enables/disables use of bullet time.
bool	`UseInGameTrajectory`	Enables/disables use of in-game trajectory rendering.
bool	`BulletTimeRunning`	Returns whether bullet time is active or not.
bool	`BulletTimeSkipPoint`	If true, bullet will travel immediately without waiting during bullet time, until set to false.
EnvironmentProperties	`GlobalEnvironmentProperties`	Returns the current environment properties assigned to this scene.

Static Variables

Type	Name	Description	Access
static float	`instance`	Singleton instance of this class.	Read-Only

Public Events

Type	Name	Description
BulletHitEvent	`EPenetrationInHit`	Called when the bullet penetrates a surface.
BulletHitEvent	`EPenetrationOutHit`	Called when the bullet exits a penetrated surface.
BulletHitEvent	`ERicochetHit`	Called when the bullet ricochets off a surface.
BulletHitEvent	`ENormalHit`	Called when the bullet hits a surface, but doesn't penetrate or ricochet.
BulletHitEvent	`EAnyHit`	Called when any type of hit occurs.
BulletTimeStartEvent	`EBulletTimeStarted`	Called when bullet time starts.
BulletTimeUpdateEvent	`EBulletTimeUpdated`	Called continuously when bullet is travelling during bullet time.
BulletTimeEndEvent	`EBulletTimePathFinished`	Called when bullet finishes it's path to it's target in bullet time.
BulletTimeEndEvent	`EBulletTimeEnded`	Called after the timescale and everything is resetted, finalizing bullet time.

BulletPropertiesEvent	`EBulletActivated`	Called when a bullet gets activated.
ZeroDistanceEvent	`EZeroDistanceChanged`	Called when zero distance is cycled.

Public Methods

Return Type	Name	Signature	Description
void	`CycleZeroDistanceUp`	()	Cycles zero distance up to the next in the m_zeroDistances list.
void	`CycleZeroDistanceDown`	()	Cycles zero distance down to the previous in the m_zeroDistances list.
void	`ActivateBullet`	(BulletProperties)	Activates the given bullet properties so that it can be used in the simulation.
void	`FireBallisticsBullet`	(BulletProperties, Transform, Transform)	Triggers the system to fire the given bullet properties, simulating it's trajectory, triggering hit & bullet time events etc.

Dynamic Scope System

Static Variables

Type	Name	Description	Access
static float	`instance`	Singleton instance of this class.	Read-Only
static float	`ScopeSensitivityMultiplier`	The sensitivity multiplier value according to the current selected zoom level.	Read-Only

Public Methods

Return Type	Name	Signature	Description
void	`ScopeActivation`	(bool, BulletProperties, float, bool)	Activates/deactivates the scope reticle, along with DSS functionality.

Player Movement Controller

Static Variables

Type	Name	Description	Access
static PlayerMotionState	`PlayerState`	Returns the current player motion state. (Idling, walking etc.)	Read-Only

Public Events

Type	Name	Description
PlayerStateActions	`EPlayerStateChanged`	Called when the player movement state changes.

Motion Controller

Public Methods

Return Type	Name	Signature	Description
void	`BreathSwayActivation`	(bool)	Activates/deactivates breath sway functionality.
void	`Recoil`	()	Recoils this object according to the recoil settings set up in the inspector.
void	`ToAimPosition`	()	Causes this object to interpolate towards the local aim position set in the inspector.
void	`ToRunPosition`	()	Causes this object to interpolate towards the local run position set in the inspector.
void	`ToHipPosition`	()	Causes this object to interpolate towards it's original position in Awake.

Ballistic Surface

Public Variables

Type	Name	Description	Access
bool	`m_penetrationEnabled`	Enables/disables penetration.	Read-Write
float	`m_penetrationEnegryConsumptionPercent`	Percent in [0,1] range defining how much of the bullet's energy will be consumed.	Read-Write
float	`m_minEnergyToPenetrateInMetrics`	Minimum energy to penetrate in metric units (Joules).	Read-Write
Vector2	`m_penetrationDeflectionAngles`	Penetrated bullet will deflect it's trajectory on x,y,z axes between this vector's x and y values.	Read-Write
bool	`m_ricochetEnabled`	Enables/disables ricochet.	Read-Write
float	`m_ricochetEnegryConsumptionPercent`	Percent in [0,1] range defining how much of the bullet's energy will be consumed.	Read-Write
float	`m_minEnergyToRicochetInMetrics`	Minimum energy to ricochet in metric units (Joules).	Read-Write

Vector2	`m_ricochetDeflectionAngles`	Ricocheted bullet will deflect it's trajectory on x,y,z axes between this vector's x and y values.	Read-Write

Environment Properties

Public Variables

Type	Name	Description	Access
float	`m_gravityInMetric`	Gravity acceleration in metric. (m/s^2)	Read-Write
float	`m_airPressureInMetric`	Air pressure in metric. (kilopascal)	Read-Write
float	`m_temperatureInMetric`	Temperature in metric. (Celsius)	Read-Write
Vector3	`m_windSpeedInMetric`	Wind vector in metric. (km/h)	Read-Write

Ballistics Utility

Public Methods

Return Type	Name	Signature	Description
Vector3	`GetWindVector`	(Vector3, float)	Calculates & returns wind vector in m/s.
Vector3	`GetGravity`	(float, float)	Calculates & returns a gravity vector.
float	`GetKineticEnergy`	(Vector3, float)	Calculates & returns a kinetic energy based on velocity & mass.
void	`UpdateKEAndVelocity`	(ref Vector3, ref float, float, float)	Updates the given kinetic energy to match the new kinetic energy, updates given the velocity accordingly as well.
Vector3	`GetDragVector`	(Vector3, DragGModel, float, float)	Calculates & returns a drag vector based on velocity, drag model & ballistic coefficient.
Vector3	`GetSpinDrift`	(ref Vector3, double, float, float)	Calculates & returns a spin drift vector based on previous drift & stability factor.
double	`GetStability`	(float...)	Calculates & returns stability factor of a bullet based on bullet properties.

double	`GetRetardation`	(double, double,DragGModel)	Calculates & returns retardation factor based on velocity, ballistic coefficient and drag model.

Bullet Time Utility

Public Static Methods

Return Type	Name	Signature	Description
Vector3	`GetRandomVector`	(Vector3, Vector3)	Returns a random vector3 between min & max.
void	`SetActualTimescale`	(float)	Changes Time.timeScale along with Time.fixedDeltaTime.
void	`ResetActualTimescale`	()	Resets Time.timeScale & Time.fixedDeltaTime.
void	`SetVirtualTimeScale`	(float)	Changes the `BulletTimeVirtualTimescale` in `SniperAndBallisticsSystem.cs`
void	`ResetVirtualTimeScale`	(float)	Resets the `BulletTimeVirtualTimescale` in `SniperAndBallisticsSystem.cs`
float	`GetRandomNegation`	()	Returns -1.0f or 1.0f based on random chance.