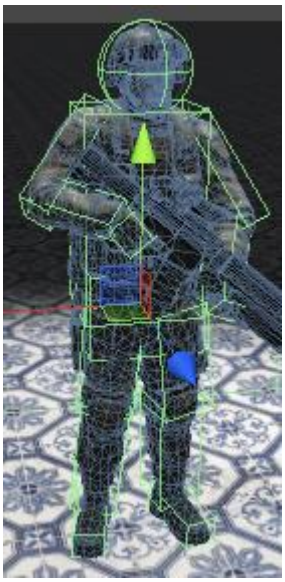
REWINDER 1.0 - MANUAL

CONTENTS

INTRODUCTION

Rewinder is a library for defining hitboxes and doing collision checks against them at different points in time. What this means in common English is that it allows you to create hitboxes, such as this:



Rewinder then records the movement of these boxes over time and allows you to cast rays and sphere overlaps against them at recorded points in time. This allows you to implement a technique called lag compensation in multiplayer games, where the server can re-wind time to compensate for the lag of players when doing hit calculations.

Rewinder also supports classifying hitboxes as different types, so you can identify when someone is shot in a specific part of the body (headshots, anyone?).

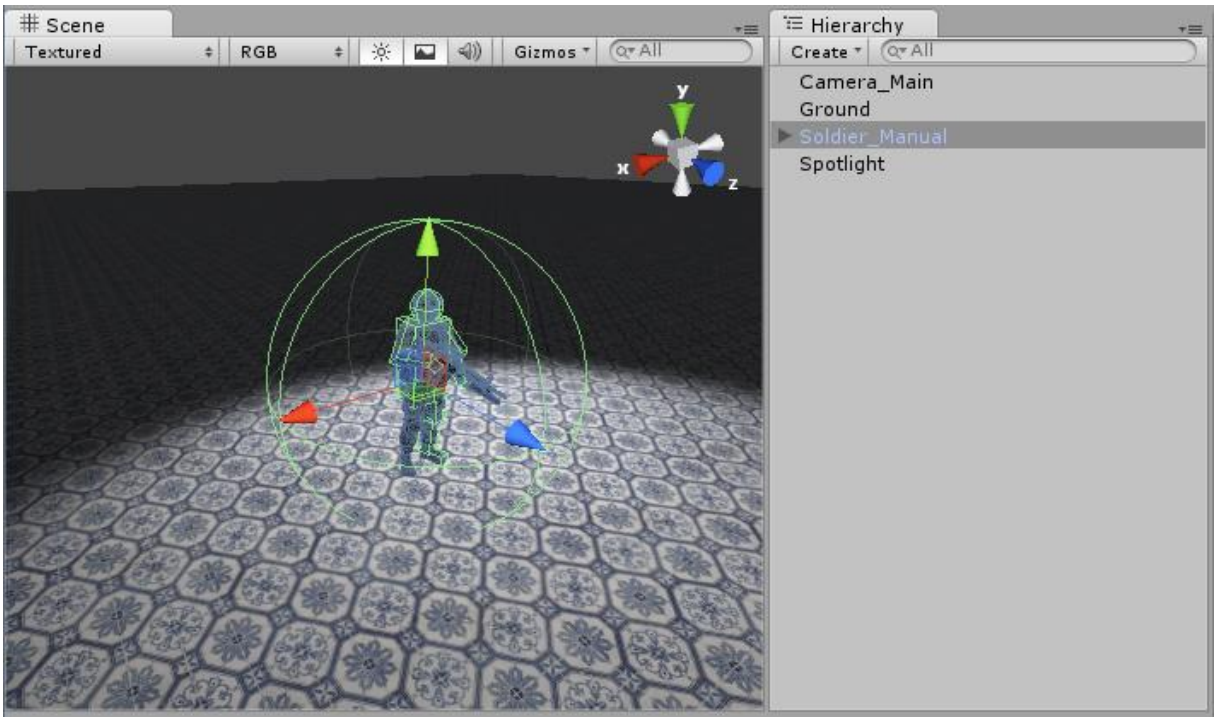
While a large part of using Rewinder is having it integrated with your networking library of choice we're going to keep it networking agnostic in this manual. However, some advice can be given on the integration.

Rewinder will create snapshots that you can access through the `RewinderSnapshot.Latest` and `RewinderSnapshot.Oldest`, each snapshot also has a pointer to the next and previous snapshot attached to it (which may be null). When sending the position and rotation from the server to the clients, send the position/rotation stored for each object in the `RewinderSnapshot.Latest` snapshot instead of reading it from their transforms, this will guarantee you the most accurate rewinding of time when doing hit calculations.

Never hold on to any references to a `RewinderSnapshot` object longer then during one frame, they are aggressively pooled and re-used by Rewinder to lower memory consumption so if you hold on to one for too long it will become pooled behind your back. If you for some reason want to turn of snapshot pooling, you can set `RewinderSnapshotPool.Instance.Enabled` to false.

CREATING THE HITBOX

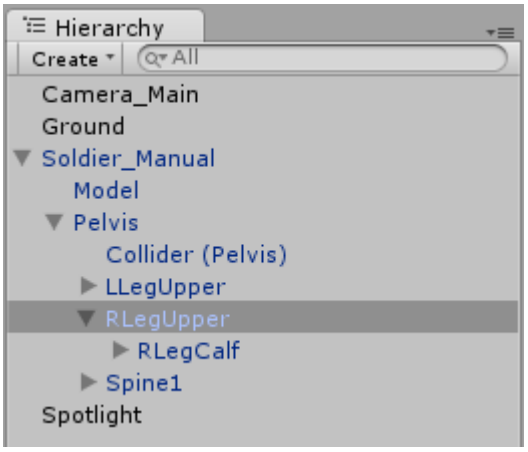
Let's dig into Rewinder, the first thing you should do is open the `Rewinder/Demo/Scenes/Manual` scene. This is an empty scene with just a basic ground, spotlight and camera prepared for you. Locate the prefab `Rewinder/Demo/Prefabs/Solider_Manual` and drag an instance of it into the scene.



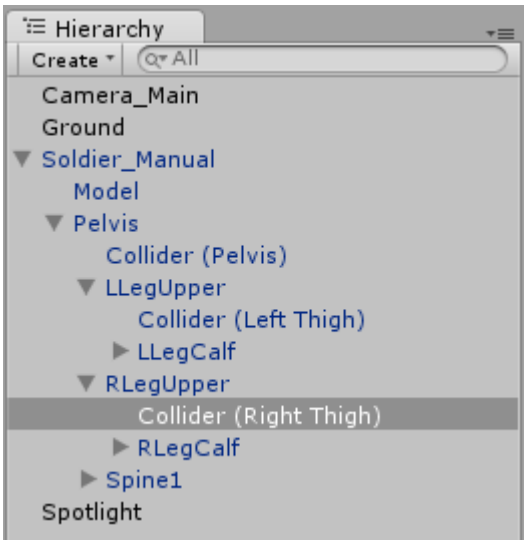
This prefab is almost completely configured; he has several colliders setup for most of his body parts, this is what Rewinder refers to as the body hitbox. He also has a big spherical collider attached that contains the entire model and some space around him; this is what Rewinder calls the proximity hitbox.

The body hitbox, which consists of several colliders, is used for identifying hits directly on a model while the proximity hitbox is used both for speeding up hit detection but also for detecting hits that are close by a player to allow you to perform different effects. Some examples of effects that can use the proximity hitbox is playing sounds of bullets flying by your head or implementing the now famous “Suppression” effect form Battlefield 3.

If you inspect the solider model closely you will see that his right leg doesn’t have any hitboxes attached to it, let’s fix this. Start by locating the `RLegUpper` bone transform:

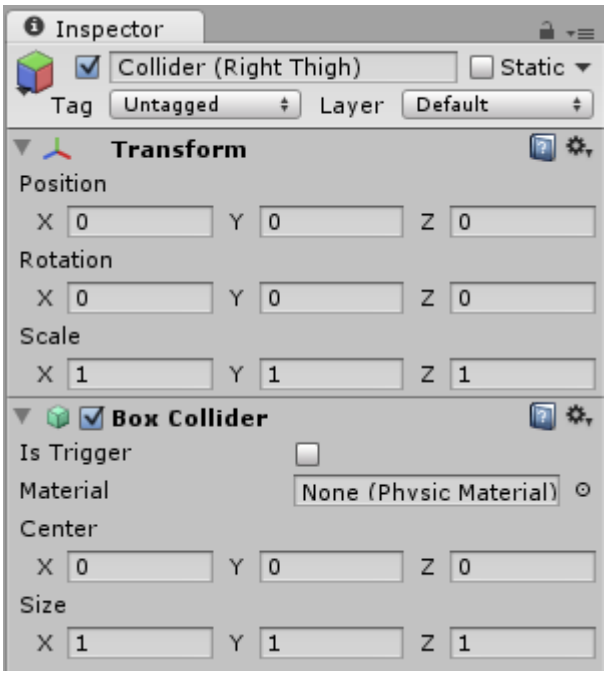


Attach a new game object as a child to `RLegUpper` and name it `Collider (Right Thigh)`.

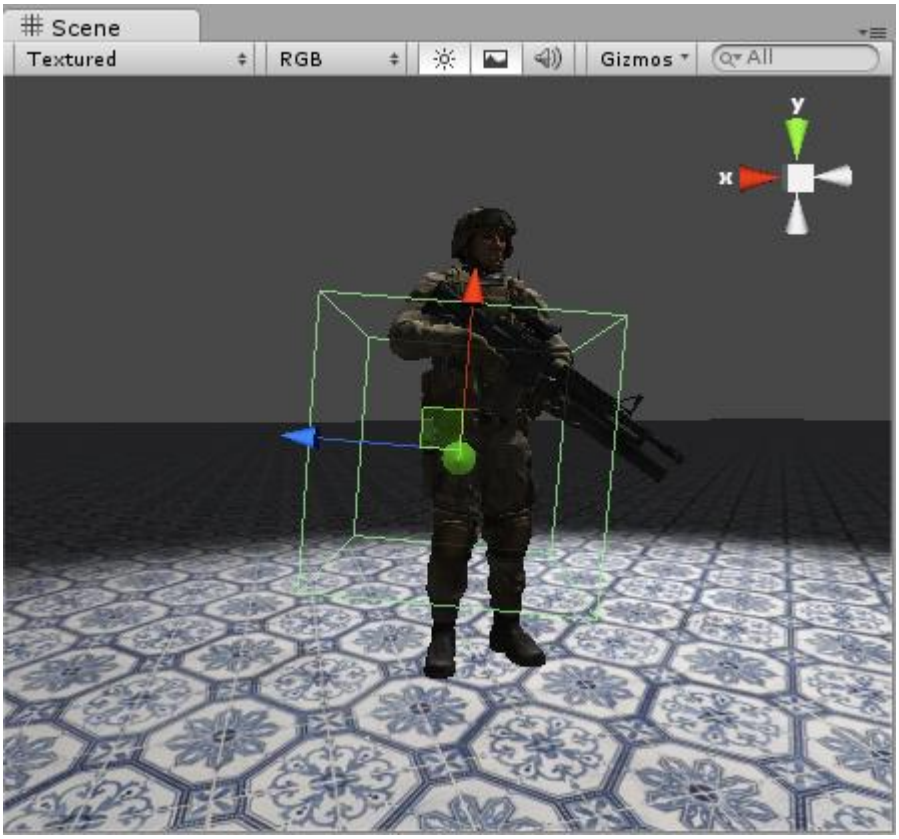


Make sure to clear the new objects position and rotation to all zeroes and its scale to all ones.

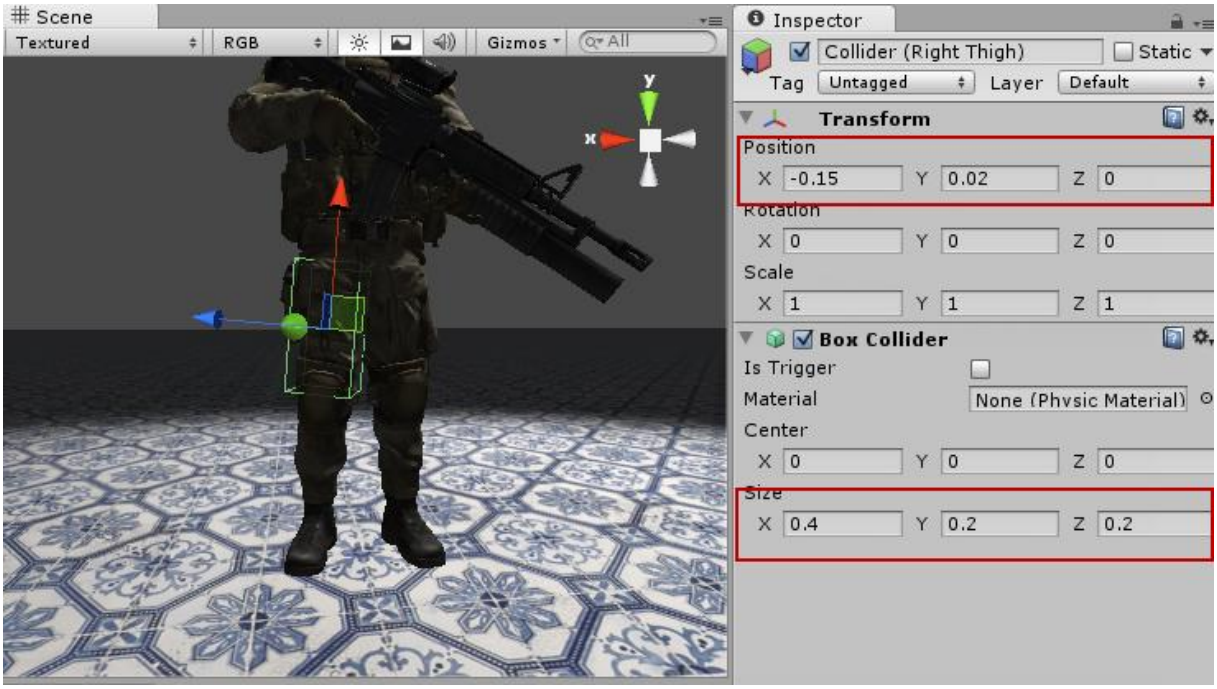
Now attach a new `Box Collider` component to it from the `Component/Physics/Box` Collider menu option.



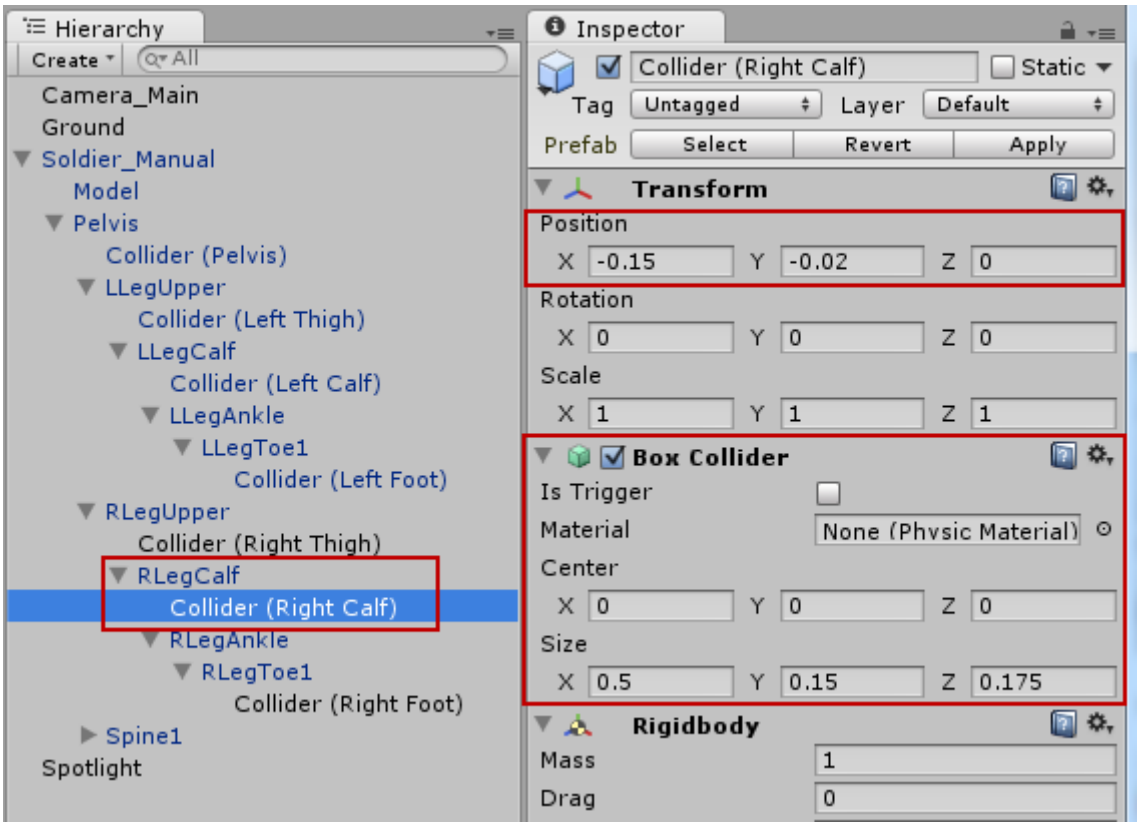
However if you look at the attached collider in the scene window, you will see that it is way too big and not really fitting the upper part of the soldiers leg.

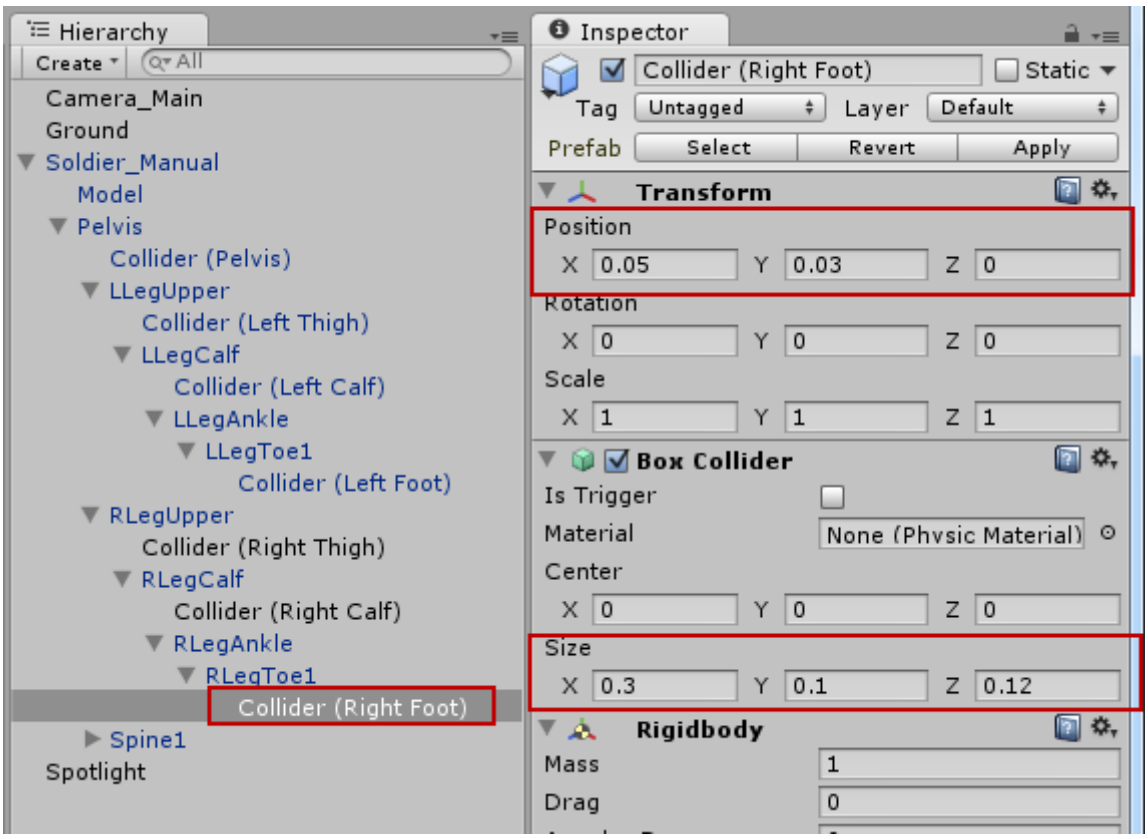


This is easily fixed, set the collider size to `0.4/0.2/0.2` and then adjust the position a bit.



Now do the same procedure for the calf and foot colliders, here are their game objects and associated settings.



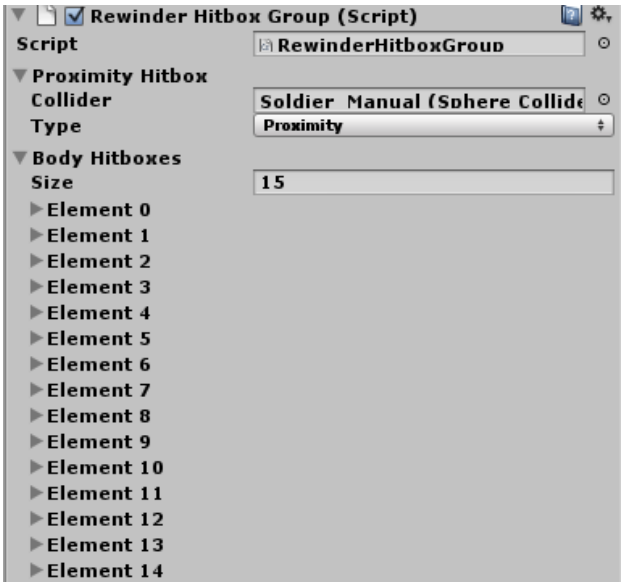


Now you have a complete hitbox on your soldier, including his right leg.

SETTING UP THE HITBOX IN REWINDER

The next thing we need to do is to setup the hitbox so Rewinder knows how to find it and track its position. Drag an instance of the `Rewinder/Scripts/RewinderHitboxGroup` script to the `Solider_Manual` instance in our scene.

There are two settings to fill in here, the proximity hitbox and the body hitbox. The easiest way is to just press “Find Colliders” and in most cases Rewinder will automatically find the correct colliders for the different hitboxes.

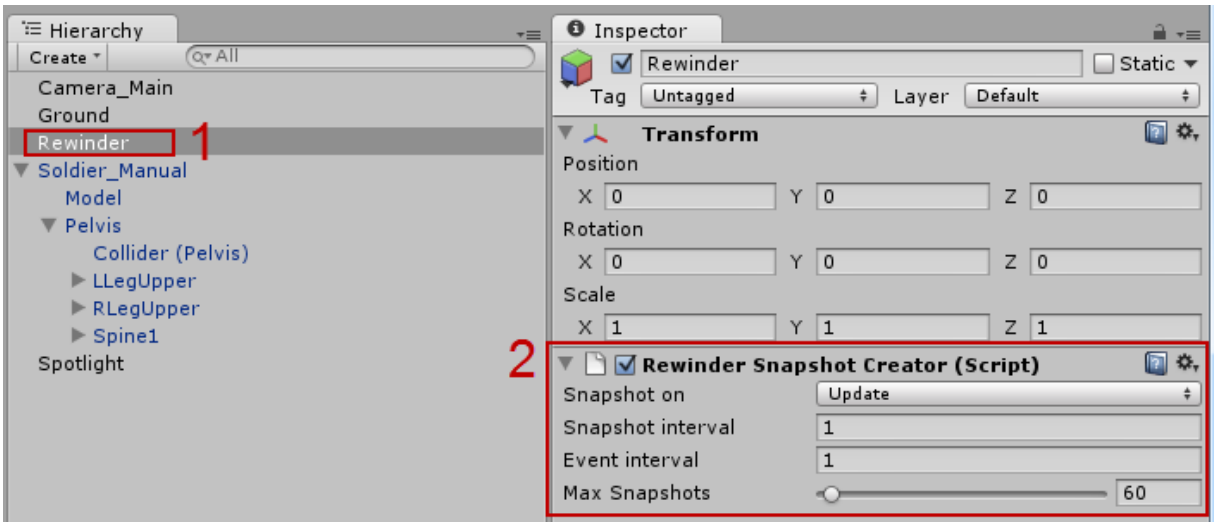


If you pressed “Find Colliders” the proximity hitbox and the 15 body hitboxes should have been found. If you expand one of the elements in the Body Hitboxes array, you can inspect what collider was found for it and also what type Rewinder identified it as.

Setting the correct Type is not required for hit detection, but it is required if you want to know what body part was hit. Rewinder tries to identify the type of the collider based on the name of the game object it’s attached to; if it can’t the default Chest type will be used.

CONFIGURING SNAPSHOT CAPTURING

The next thing we’re doing is configuring Rewinder to actually capture snapshots of our world at regular intervals, the most basic way of capturing a snapshot is just to call `RewinderSnapshot.Capture()`, which will return a snapshot object which represent the current state of all hitbox groups tracked. If you need completely custom integration with Rewinder calling `Capture` allows you the most freedom to do exactly what you want. For most needs using the pre-built `Rewinder/Scripts/RewinderSnapshotCreator` script attached to a game object somewhere in the scene is more than good enough. This is exactly what we’re going to do. Create a new game object and call it `Rewinder`, then drag an instance of the `RewinderSnapshotCreator` script to it.



- **Snapshot on** – If you want to create a snapshot of the hitbox states after a complete Update cycle or Fixed Update cycle. This basically decides if the snapshots are frame bound or physics bound.
- **Snapshot interval** – How often we should create a snapshot, 1 means every frame, 2 means every other frame, etc.
- **Event interval** – There is a public event called `NewSnapshot` available on the `RewinderSnapshotCreator` class, which can be reached through `RewinderSnapshotCreator.Instance`, this property controls how often this even should get raised. One means every snapshot; two means every two snapshots, etc.
- **Max Snapshots** – the max amount of snapshots allowed to be stored in memory, I usually set this to 1/expected frame time, or 1/fixed time step, depending on if you snapshot on rendered frames or physics updates.

The default values works very well, except for the Event interval which I usually set to 3. You generally do not need more than one second (60 frames, assuming 60 FPS) of snapshots saved to do proper lag compensation; if someone has a ping that is higher than ~1000ms chances are the game is not playable anyway, in summary:

- Snapshot on: Update
- Snapshot interval: 1
- Event interval: 3
- Max snapshots: 60

Why does this work so well? If you snapshot every frame at 60 fps (~16.67ms per frame) this will give you an average accuracy of ~8.33ms (half the time of one frame), which is far lower than any person can perceive. Generally you will send about 20 updates to the clients per second, and 60 / 3 (event interval) = 20. So you will get a good accuracy when you do hit detection and a nice smooth state rate to the clients.

Now, generally you would attach an event listener to the previously mentioned `NewSnapshot` event, and every time the event gets triggered you'd collect the position and rotation data from the snapshot and distribute it to the clients.

Exactly how you transmit the snapshot data to the clients depends on your networking solution; Rewinder itself does not enforce any specific way of transferring the data. However, you might want to look at the `RewinderSnapshot.GetTime` delegate, it's used by Rewinder to get the timestamp for the snapshots. The default one returns the value of `UnityEngine.Time.time`, but if you're using a third party networking solution such as Lidgren, uLink or SlimNet they may have their own timing algorithm which will be synchronized between the server and clients. If this is the case it's recommended that you replace the `RewinderSnapshot.GetTime` delegate with a function that returns this value.

CASTING RAYS AND SPHERES

When everything is up and running the API itself is extremely simple, you have two static methods to the `RewindSnapshot` class which is your bread and butter:

- `bool Raycast(float time, Vector3 origin, Vector3 direction, out List<RewinderHitboxHit> result)`
- `bool OverlapSphere(float time, Vector3 origin, float radius, out List<RewinderHitboxHit> result)`

The only major difference compared to the methods available in `UnityEngine.Physics` is that these two take a time parameter as their first argument. The time parameter controls how far Rewinder will try to rewind time and perform the hit check, it will only rewind time as far as the oldest snapshot saved – and if you go further back then this it will fall back to the oldest one. If for example you would like to cast a ray 100 milliseconds behind current time, it would look like this:

```
List<RewinderHitboxHit> result;

if(RewinderSnapshot.Raycast(0.1f, Vector3.zero, Vector3.forward, out result)) {
    // We hit something
} else {
    // We did not hit anything
}

RewinderSnapshot.Recycle(result);
```

Now you might ask what the call to `RewinderSnapshot.Recycle` is at the end, it's to lessen the pressure on the garbage collector, when you're done with a result list you can pass it back to Rewinder and it will be re-used, this is a lot more efficient then constantly creating new lists.

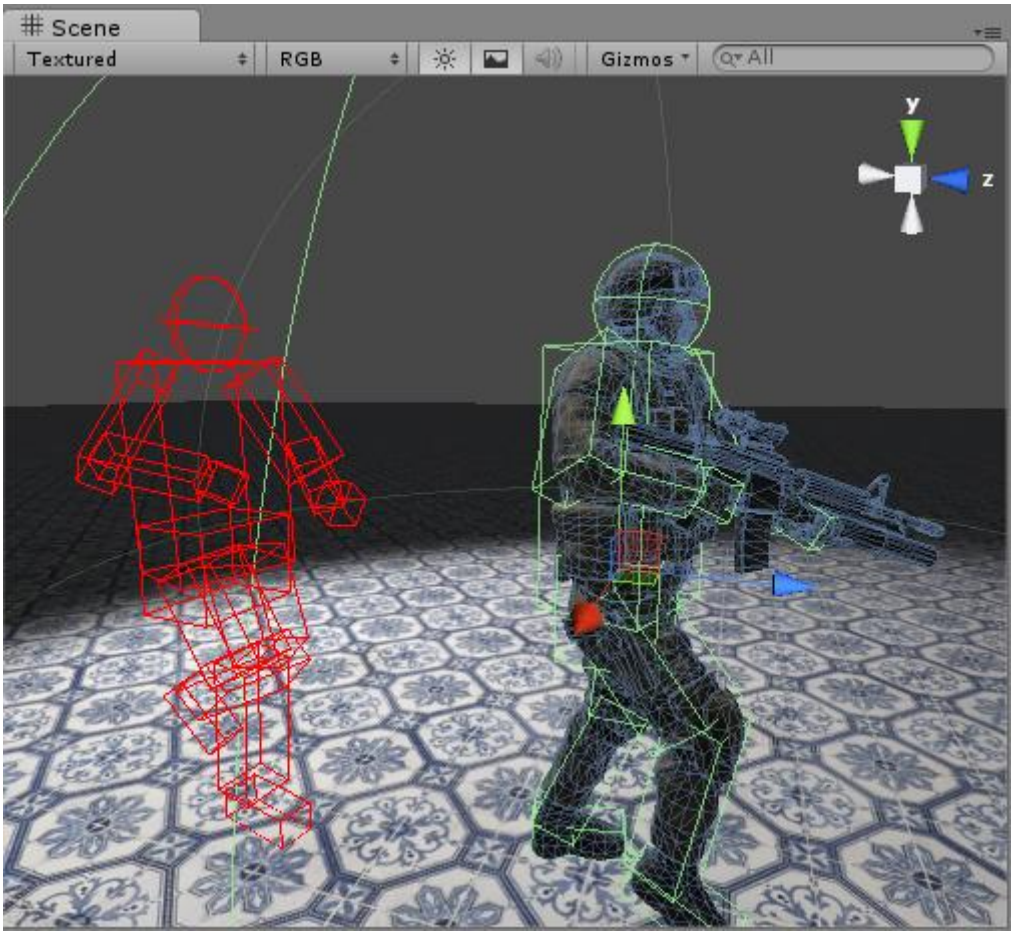
The result passed back from the `Raycast` method is a list of `RewinderHitboxHit` values, each one contains three values:

- **Distance** – the distance between the object hit and the origin of the ray cast, this is -1f for sphere overlaps.
- **Group** – Which hitbox group was hit, through the group you can find the transform of the game object hit.
- **Hitbox** – An instance of the RewinderHitbox class which contains information about the hitbox that was hit, for example if it was in a Head or Chest box, etc.

And that is really all there is to casting rays and spheres using Rewinder, there are a couple of other variations of the `Raycast` and `OverlapSphere` methods which allow you to pass in your own list that will get populated, but that's about it.

DISPLAYING DEBUG INFORMATION

If you need to display debug information about a snapshot, every `RewinderSnapshot` instance has a `DrawGizmos` method on it that can be used to draw the hitboxes (in red) at the position they are in the snapshot.



If you're calling `DrawGizmos` but not seeing anything, verify three things:

- You're running inside the editor
- It's being called from a `OnDrawGizmos` method
- The `#define REWINDER_DEBUG` statement at the top of the `Rewinder/Scripts/RewinderHitboxGroupSnapshot.cs` file is not commented out.