Introduction

Bullet Ballistics combines the **accuracy** of hit-scan based systems, with the **realistic bullet drop** of rigidbody based projectiles, and additionally includes **air resistance** in its ballistics simulation.

It has been built from the ground up with **performance** in mind, making full use of **multithreading** utilizing unity's job system and burst compiler. While also minimizing runtime garbage collection allocations wherever possible.

Another main development goal was creating **customizable** interfaces to the core ballistics simulation, allowing for custom bullet rendering, material interactions, and impact handling.

All together, Bullet Ballistics empowers you to simulate hundreds to thousands of physically accurate projectiles in your games simultaneously without a hustle.

Links

- Online Demo
- Documentation
- Video Tutorials
- E-Mail: mr3d.cs@gmail.com

Required Unity Packages

- `Unity.Burst`
- `Unity.Mathematics`

Features

- Ballistics Simulation
  - Gravity / Bullet Drop
  - Air Resistance
  - Wind
  - Weapon Zeroing + Reticle Generation
- Material Interactions
  - Object Penetration
  - Ricochets

- High Performance
    - Multi-Threading
    - Minimal Runtime GC Allocations
    - Batched Projectile Rendering
    - Object Pooling
- Customizability
    - Bullet Rendering
    - Impact Handling
    - Material Interactions
- Clear Custom Inspectors
    - Physical Unit Selection (metric/imperial)

Technical Details

The ballistics simulation is completely independent of the visual projectiles in the game. This allows for a lot of customizability in the way the bullets are rendered: basic prefab instances, GPU instancing, custom renderer? Your choice! This also makes Bullet Ballistics inherently independent of the Rendering Pipeline you target.

Internally, Bullet Ballistics uses a numerical approximation to simulate the ballistic trajectory. For collision detection, a raycast is fired between the last two simulated positions of a given projectile. All ballistics processing is automatically batched using unity's RaycastCommand API and job system for optimal performance.

Limitations

Bullet Ballistics does (currently) not support ballistic effects like spin drift, coriolis force, magnus effect, or similar, as they only affect very long range projectiles, and are usually negligible for normal game scenarios. Feel free to contact me with any feature requests!

Getting Started

Check out the included example scene in the `Ballistics/Example/` folder.

Minimal Setup

- create a `Assets/Create/Ballistics/Ballistic Settings` object in your project folder
- add an empty game object to your scene and attach a `Environment Provider` component to it
  - assign the `Ballistic Settings` object you have created before
- attach a `Weapon` component to a weapon in your scene
  - assign a transform at the end of the weapon barrel as the `Bullet Spawn Point` (z-axis pointing forward)
  - create a `Assets/Create/Ballistics/Bullet Info` object in your project folder
  - assign this `Bullet Info` object in the `Weapon` component
- calling `Shoot()` on the `Weapon` component will fire a bullet (you can use the `Simple Weapon Input` component, to shoot when clicking the left mouse button)

Note! As so far no impact handler or visual bullet provider is set up, you will *NOT* see the bullet or any impacts!

Visual Bullet Provider

Bullet Ballistics separates the ballistics simulation from the visual projectiles flying through the scene. This allows to optimize the rendering when thousands of projectiles are active simultaneously, and allows you to customize the rendering to your needs.

- create a `Assets/Create/Ballistics/Visual Bullet Providers/Pooled Bullet Provider` object, and assign a bullet prefab with a `Pooled Bullet` component attached
- assign the new `Pooled Bullet Provider` object to the `Visual Bullet Provider` field of your weapon

You should now be able to see the bullet prefab when shooting the weapon.

Impact Handler

Similar to the `Visual Bullet Provider` the `Impact Handler` allows you to customize what should happen on a projectile impact.

- create a `Assets/Create/Ballistics/Impact Handler/Generic Impact Handler` object in your project folder
  - assign the impact handler object to the `Global Impact Handler` field of your `Environment Provider` component
- enabling `Handle Rigidbody Forces` should cause rigidbodies to be affected by projectile impacts
- adding a audio clip to the `Impact Sounds` list, will cause an impact sound at any impact of a projectile with the scene

Ballistic Materials

Ballistic Material define how a projectile will behave when interacting with a collider.

- create a `Assets/Create/Ballistic/Ballistic Material` object in your project folder
  - a Ballistic Material is a unity `Physic Material` with an associated `Ballistic Material` object
  - you can promote/demote a `Physic Material` to/from a `Ballistic Material` by right-clicking the `Physic Material` and selecting `Assets/Create/Ballistics/To BallisticMaterial` / `Unlink BallisticMaterial`
- assign a Ballistic Material to a collider by setting the `Material` field of the collider to a `Physic Material` associated with a `Ballistic Material` object
- by change the properties of the Ballistic Material to alter the bullet interactions

For more information on each of these topics check out the following chapters.

Visual Bullet Provider

A `Visual Bullet Provider` is just a scriptable object, with a `GetVisualBullet()` method, which returns a new `IVisualBullet`.

```
public abstract class VisualBulletProviderObject :
InitializableScriptableObject
{
    public abstract IVisualBullet GetVisualBullet();
}
```

The `IVisualBullet` interface consists of:

```
public interface IVisualBullet
{
    void InitializeBullet(in BulletPose pose, in float3 visualOffset);  //
start of ballistic simulation
    void UpdateBullet(in BulletPose pose);                              // pose
update, each simulation cycle
    void DestroyBullet();                                               //
bullet has been destroyed (stopped or timeout)
}
```

The `visualOffset` argument of `InitializeBullet()` will be explained further in the chapter about the Weapon component.

The `Visual Bullet Provider` adds a layer of abstraction between the simulation and the rendering. This allows switching between rendering strategies easily.

For example, the `Pooled Bullet Provider` uses basic prefab instanciation for rendering the visual bullet (+ pooling instead of destroying bullets, to minimize garbage collection overhead).

Switching the visual bullet of a `Weapon` to the `Tracer Bullet Provider` can be done, by simply replacing the `Visual Bullet Provider` of the `Weapon` component. The `Tracer Bullet Provider` draws bullets, by rendering a trail between the last three bullet positions. The `Tracer Bullet Provider` can render bullets very efficiently, as it is able to combine ~10.000 bullets into a single draw call.

With this flexibility it is very easy to create your own `Visual Bullet Provider`s which fit your game's requirements optimally!

Impact Handler

Similar to a `Visual Bullet Provider` a `Impact Handler` is also a scriptable object, which adds a layer of abstraction between the ballistic simulation and the rest of your game.

```
public abstract class ImpactHandlerObject : InitializableScriptableObject,
IImpactHandler
{
    public abstract HandledFlags HandleSurfaceInteraction(in
SurfaceInteractionInfo impact, HandledFlags flags);
    public abstract HandledFlags HandleImpact(in ImpactInfo impact,
HandledFlags flags);
}
```

- `HandleSurfaceInteraction` is called whenever a projectile first enters, ricochets, exits, or stops inside of a collider
- `HandleImpact` is called when a projectile penetrates a collider

Similar to the `Pooled Bullet Provider` the `Pooled Impact Instantiator` instantiates a prefab with an attached `Impact Object` component at a surface interaction (and pooles the instantiated prefabs).

Alternatively, the `Generic Impact Handler` provides default handling for rigidbody interactions, a damage system, and audio effects.

Chain Of Responsibility

As some bullet interactions might be specific to the bullet, material, or collider, the `Impact Handler` system is structured hierarchically.

The `Bullet Info`, `Ballistic Material`, `Environemnt Provider`, and collider can each provide an `Impact Handler`. When a bullet hits a collider the impact handlers are called (if present) in this order:

- `Bullet Info` specific `Impact Handler`
- `Ballistic Material` specific `Impact Handler`
- global impact handler provided by the `Environment Provider`

The collider specific `Impact Handler` is not called by default, as this requires a `GetComponent` call! You can query for a collider specific `Impact Handler` inside any of the above impact handling stages, by trying to get a component inheriting the `IImpactHandler` interface and calling the specific handler.

When collider specific a `Impact Handler` is needed globally, you can use a `Generic Impact Handler` as the global `Impact Handler`, and enable `CheckForPerColliderHandler`.

At each step of the `Impact Handler` chain you receive the `HandledFlags` of the previous handlers, and can return which flags have been handled by the current handler. This allows, for example, a bullet impact handler to override the bullet hole prefab of the global impact handler, or a material impact handler to alter the impact sound effect.

Impact Info

```
public readonly struct SurfaceInteractionInfo
{
    public enum InteractionType
    {
        STOP,
        ENTER,
        RICOCHET,
        EXIT,
    }
    public readonly InteractionType Type;
    public readonly RaycastHit HitInfo;
    public readonly float3 Velocity;
    public readonly float SpreadAngle;
    public readonly float SpeedFactor;      // percentage of speed lost on
ricochet
    public readonly BulletInfo BulletInfo;
    public readonly Weapon Weapon;
}

public readonly struct ImpactInfo
{
    public readonly RaycastHit HitInfo;
    public readonly float3 EntryVelocity;
    public readonly float ImpactDepth;
    public readonly float EnergyLossPerUnit;
    public readonly BulletInfo BulletInfo;
    public readonly Weapon Weapon;
}
```

Environment Provider

Each scene using Bullet Ballistics should have a single game object with a `Environment Provider` component attached.

The `Environment Provider` configures the ballistic simulation setting for the current scene. Including:

- Ballistic Settings
- Wind Velocity
- Global Impact Handler
- Bullet Path Debugging (Editor only)

In theory these settings can also be adjusted manually in the static `Ballistics.Core` class. The `Environment Provider` automatically configures the `Ballistics.Core` on startup/ scene change.

The fields of the `Environment Provider` will be explained further in the respective chapters.

Weapon

The `Weapon` component is usually the main entry-point for interacting with the ballistics simulation.

It is mainly just a convenience component, that allows you to combine a `Bullet Info`, `Visual Bullet Provider`, and spawn point transform, so you only have to call `Weapon.Shoot()` to fire a bullet.

Visual vs. Bullet Spawn Point

The `Weapon` component exposes two spawn point transforms, the `Bullet Spawn Point` and `Visual Spawn Point`.

- `Bullet Spawn Point` is the position used for the internal ballistics simulation and collision detection (z-axis of the transform pointing forward)
- `Visual Spawn Point` provides the initial position for the `Visual Bullet` (if present)

This distinction is often useful for first person shooter games, because the hit detection should usually originate from the center of the camera, while the visual bullet representation should appear to originate at the end of the weapon's barrel.

This obviously causes a discrepancy between the visual bullet and the internal hit detection. To mitigate this offset, Bullet Ballistics automatically moves the visual bullet position towards the internal bullet position over a given distance set in the `BallisticSettings` `Visual To Physical Distance`.

Manually Queuing Bullets

Internally, `Weapon.Shoot()` calls `Ballistics.Core.AddBullet()` with a new `BulletInstance`.

```
public readonly struct BulletInstance
{
    public readonly Weapon Weapon;
    public readonly Vector3 Position;
    public readonly Vector3 Direction;
    public readonly BulletInfo Info;
    public readonly IVisualBullet Projectile;
}
```

You can also manually add bullets to the ballistics simulation using `Ballistics.Core.AddBullet()` at any point. Note however, that `AddBullet()` just queues bullets for processing, so the ballistics calculations can be batched together with all other active bullets.

Queued bullet instances are added to the simulation just after the main `Update()` loop. So all bullets queued after `Update()` (e.g. `LateUpdate()`) will not be processed until the next frame.

Zeroing Crosshair Generator

Due to bullet drop when gravity is enabled, bullets will hit below where you have aimed. To correct for this, you have to aim above your target.

Given a `Weapon` and a set of distances, this component will generate and draw reticle 'dots' for hitting targets at the given distances.

The other fields of this component allow you to specify the appearance of the generated indicator mesh. You can use this generated mesh directly, or use it as a template for drawing a custom reticle with the proper offsets in an image creation program of your choice.

Weapon Controller

The `Weapon Controller` component is a convenience wrapper around a basic `Weapon` component.

It allows your `Weapon` to behave like one of the most common weapon types (`Full Auto` , `Single Shot` , `Shotgun` , or `Burst`). You only have to pull/release the trigger using:

```
WeaponController.SetTrigger(bool held);
```

It also allows you to add basic spread and bullet management.

Zeroing

The `Weapon Controller` also includes zeroing the weapon, to counteract bullet drop caused by gravity for a given distance.

Specify the `Distances` you want to be able to zero your weapon at, and set the `CurrentZeroingIndex` . **Note!** when updating the `Weapon` / `Bullet Info` or `Distances` at runtime, you have to call `UpdateZeroing()` , to recalculate the zeroing angles.

The `Weapon Controller` simply angles the bullet's shoot direction slightly upwards to hit targets further away.

Spread Controller

```
public class SpreadController : MonoBehaviour
{
    public virtual Vector3 CalculateShootDirection(Weapon weapon) { return
weapon.BulletSpawnPoint.forward; }
    public virtual void BulletFired() { }
}
```

`CalculateShootDirection` is called for every bullet fired by the weapon and specifies in which direction it is fired. `BulletFired` is called by the `WeaponController` each time is shoots (regardless of the number of bullets fired in the single shot). This can be used for e.g. increasing the spread angle of following bullets.

As an example for a custom spray controller, `Default Spread Controller` implements a Counter-Strike-style spray pattern controller, where you can specify the spray pattern via unity's `Animation Curve` s.

Magazine Controller

```
public class MagazineController : MonoBehaviour
{
    public virtual bool IsBulletAvailable() { return true; }
    public virtual void BulletFired() { }
}
```

The `Magazine Controller` is queried each time before the `Weapon Controller` is fired, to check if `IsBulletAvailable()`.

`DefaultMagazineController` implements a very basic magazine based bullet management.

Ballistic Material

A `Ballistic Material` specifies how a projectile interacts with a collider.

Custom `Ballistic Materials` can be created by implementing the `IBallisticMaterial` interface, and associating them with a `Physic Material` at runtime, by adding them to the `BallisticMaterialCache`.

However, for must cases it will be sufficient to create a `Ballistic Material` scriptable object via `Assets/Create/Ballistics/Ballistic Material`. These will be associated with a `Physic Material` automatically.

```
public interface IBallisticMaterial
{
    MaterialImpact HandleImpact(in float3 velocity, BulletInfo info, in
RaycastHit rayHit);
    float GetEnergyLossPerUnit();
    float GetSpreadAngle();
    IImpactHandler GetImpactHandler();
}
```

Combining `Ballistic Material`s with unity's `Physic Material`s has the advantage that unity directly provides the `Physic Material` of the hit collider from a raycast query, so no `GetComponent` call to something like a *`Ballistic Material Provider`* component on the object is required, which would come with a performance impact.

The main method of the `IBallisticMaterial` interface is `HandleImpact`. It provides all the available information about the impact, and the `Ballistic Material` decides how the projectile should continue, by returning a `MaterialImpact` struct.

```
public readonly struct MaterialImpact
{
    public enum Result
    {
        STOP,
        ENTER,
        RICOCHET
    }
    public readonly Result ImpactResult;
    public readonly float SpreadAngle;
    public readonly float SpeedFactor;
}
```

The bullet can either be stopped right away, be reflected (ricochet), or penetrate (enter) the collider. The `SpeedFactor` is multiplied with the current bullet velocity to result in the new bullet velocity.

The `SpreadAngle` only applies to `Ricochet` bullets. When a bullet penetrates a collider, the spread is applied on the exit and is determined by the `GetSpreadAngle` method.

For additional information about the `Ballistic Material` object fields, please refer to the inspector tooltips.

Ballistic Material Browser

For an easy overview over all `Ballistic Material`s in your project, open the `Window/Ballistics/Ballistic Material Browser` editor window.

Ballistic Settings

The `Ballistic Settings` object exposes parameters for adjusting the internal ballistics simulation. To bind a `Ballistic Settings` object to a given scene, use an `Environment Provider` component.

Create a new `Ballistic Settings` object under `Assets/Create/Ballistics/Ballistic Settings`.

Alternatively, you have direct access to the simulations settings via the static `Ballistics.Core` class. Make sure you call `Ballistics.Core.UpdateEnvironment()`, after changing the simulation settings, to rebuild the `Ballistics.Environment` used by the internal simulation jobs.

For additional information about the `Ballistic Settings` object fields, please refer to the inspector tooltips.

Bullet Info

The `Bullet Info` object allows setting physical information about the bullet fired by a weapon, which layers the bullet is able to hit, and a bullet specific `Impact Handler` object.

Create a new `Bullet Info` object under `Assets/Create/Ballistics/Bullet Info`.

For additional information about the `Bullet Info` object fields, please refer to the inspector tooltips.

Contact

Any questions or suggestions?

- Christian Schott
- E-Mail: mr3d.cs@gmail.com
- YouTube: https://www.youtube.com/channel/UC2-tdH9kaJTJ36vbzB31fbg
- Unity AssetStore: https://assetstore.unity.com/publishers/16528