# VariaBULLET2D - Scripting Guide

by Neon Dagger

# 1. Pattern Scripts

A collection of scripts which directly result in setting up and automating emitter patterns & controller states.

- **AutomateBase.cs** – base class for Automators.
- **AutomateLinear.c**s – Linear Automator class for automating controllers.
- **AutomateStepped.cs** – Stepped Automator class for automating controllers.
- **BasePattern.cs** – base class for SpreadPattern monobehavior.
- **SpreadPattern.cs** – monobehavior class which drives controller behavior.
- **BasicPresetState.cs** – class which can hold preset state for controller presets.
- **FullPresetSwitcher.cs** – monobehavior for enabling live switching between controller presets.

# 2. Shot Scripts

A collection of scripts which drive bullet and laser shot behaviors.

- **BlastAnimBase** – base class for creating blast behavior.
- **BlaseMissile** – class for creating a simple blasting jet animation seen in homing missiles.
- **FireBase.cs** – base monobehavior class for firing script attached to emitters.
- **FireBullet.cs** – monobehavior firing script attached to emitter for firing bullet type shots.
- **FireExpanding.cs** – monobehavior firing script attached to emitter for firing laser type shots.
- **GlobalShotManager.cs** – singleton monobehavior class which instantiates the GlobalShotManager prefab used to manage a host of global game states.
- **ShotBase.cs** – base monobehavior class for handling all basic shot behaviors.
- **ShotBaseAnimatable.cs** – monobehavior for Shots that can be animated with sprite frames.
- **ShotBaseRotatable.cs** – monobehavior for Shots that can be rotated.
- **ShotBaseColorizable.cs** – monobehavior for Shots that can be colorized in real-time.
- **ShotBullet.cs** – base monobehavior class for all bullet type Shots.
- **ShotAccelerating.cs** – monobehavior which propels Shots at an accelerating pace.
- **ShotBoomerang.cs** – monobehavior for Shots which come back to their emitter after a set time limit.
- **ShotBurstPhysics.cs** – monobehavior for Shots using inertial physics in bursting intervals.
- **ShotEmitPattern.cs** – monobehavior for Shots which carry their own controllers/emitters that can fire their patterns during the parent Shot’s trajectory.
- **ShotExploding.cs** – monobehavior for Shots which carry their own controllers.emitters, setting off an event that appears to explode the bullet into more bullets.
- **ShotParticleExplodeOnHit.cs** – similar in structure to ShotExploding, however, triggers the appearance of an exploding bullet once it collides with another object.
- **ShotGravitational.cs** – monobehavior for physics Shots which respond to gravity, particularly arcing trajectories.
- **ShotHoming.cs** – monobehavior for Shots which home in on targets, either singular or in groups.
- **ShotHomingInertial.cs** – monobehavior for Shots which home in on targets, either singular or in groups, using inertial bursts in intervals.
- **ShotLaser.cs** – monobehavior base class for all laser type Shots.
- **ShotLaserPacket.cs** – monobehavior class for lasers which detach into bullet type packets.
- **ShotLinearNonPhysics.cs** – monobehavior class for all linear non-physics Shots.
- **ShotNonPhysics.cs** – repoolable version of ShotLinearNonPhysics.
- **ShotLinearPhysics.cs** – monobehavior class for all repoolable linear physics-based Shots.
- **ShotPhysics.cs** – base monobehavior class for physics-based Shots.
- **ShotReAngle.cs** – monobehaviour class for Shots which change their trajectory at an angle.
- **ShotSpeedWave.cs** – monobehavior class for Shots which change their acceleration in a wave-like pattern.

## 3. Collision Scripts

A collection of scripts which are used for determine collisions and damage.

- **CollisionFilter.cs** – static class with helper methods for filtering acceptable collision objects.
- **IDamager.cs** – interface used by shots in order to apply damage in scripts that receive incoming collisions.
- **ShotCollidable.cs**:
  - **IShotCollidable** – interface used by objects that receive shot collisions.
  - **CollisionArgs** – class used to pass arguments in a single object to OnLaserCollision().
- **ShotCollision.cs** – monobehavior class attached to objects which receive shot collisions.
- **ShotCollisionDamage.cs** – monobehavior class attached to objects which receive shot collisions as well as damage (player, enemies, destroyable objects, etc).
- **Explosion.cs** – Monobehavior driving explosion animations.
- **DamagerBody.cs** – Monobehavior that can be attached to damaging objects (player, enemy, spikes, etc) in order to deal damage to other objects it collides with, in the same way that shots do.

## 4. Utility Scripts

Scripts which contain functionality common to a range of other scripts within the system.

- **BasicAnimation.cs** – class which handles sprite frames in order to create simple animations, particularly in the case of animatable bullets.
- **ForceGlobalShotManager.cs** – monobehavior for forcing instantiation of the GlobalShotManager at the beginning of runtime. Typically attached to the persistent Camera GameObject.
- **HomingCalc.cs** – class for handling helper routines relevent to Homing Shot types.
- **ObjectPool.cs** – class for managing pooled objects, typically bullets or explosions.
- **OnScreenDisplay.cs** – monobehavior for displaying stats via the GlobalShotManager.
- **Timer.cs** – class which forms internal counter/timer behavior used by many scripts.
- **Utility.cs**:
  - **Utilities** – static class with helper methods for debugging.
  - **CalcObject** – static class with helper methods for common calculations required for a variety of shots.
- **VariaManager.cs** – static class for creating the VariaManager menu item in the editor and defining the list of file management procedures.

## 5. Editor Scripts

Scripts used to add inspector functionality to other scripts.

- **BasePatternEditor.cs** – class for creating Automator buttons in controller BasePattern scripts.
- **FireBaseEditor.cs** – class for creating Audio Event button in emitter FireBase firing scripts.

## 6. Demo Scripts

A variety of scripts not crucial to VariaBULLET2D as a system, but used to drive behavior in included demo scenes. See script synopsis at the top of each script file for a broader description.

## 7. Custom Shot Scripts

At some point you will likely want to create your own custom Shot scripts. The entirety of VariaBULLET2D, including shot behaviors, is driven by the underlying Unity system of components and monobehaviors and is therefore highly extensible, allowing you to create any behavior you wish. This guide will cover some basic rules of thumb to keep in mind in order to easily create your own custom behavior.

## 7.1. Shot Inheritance

Since VariaBULLET2D shots have an existing inheritance scheme, you will want to choose an existing shot class that your custom script inherits from, and extend new behavior from there. So, the first step is determining which shot class houses the most fundamental functionality that's closest to the behavior you want to create.

The following is an inheritance tree of existing shot classes. For better definitions of these classes, see the Shot Scripts(http://neondagger.com/variabullet2d-scripting-guide/#shot-scripts)
section above.

- **ShotBase**
    - **ShotLaser**
    - **ShotBullet**
        - **ShotBaseAnimatable**
            - **ShotBaseRotatable**
                - **ShotBaseColorizable**
                    - **ShotLinearNonPhysics**
                    - **ShotLinearPhysics**

## 7.2. Shot Methods & Events

**ShotBase Class** – as you can see from the inheritance tree above, the ShotBase class is the most fundamental class to all shot behaviors. When creating your own class, it's important to understand a few of the key methods from this class in order to make the most of your custom variant.

- **InitialSet()** – this is the first method called when a firing script instantiates a shot and as such becomes a sort of pseudo-constructor. Override to implement custom initializations you might need for your shot. Note that component properties via GetComponent may not be available.
- **Start()** – marked as virtual so it must be overriden if used in subclasses. Might be necessary when a component property is required but not available via InitialSet().
- **Update() / LateUpdate()** – typical update methods. Can be overriden in subclasses.
- **OnStoppedFiring()** – an event which performs an action when the controller that the shot came from stops firing.
- **OnEmitterDestroyedDo()** – call this method and input a custom callback method which will be called on every frame when the emitter that fired this shot becomes destroyed.
- **OnEmitterDestroyedDoOnce()** – same as OnEmitterDestroyedDo except it only fires on one frame upon emitter destruction.
- **OnEventTimerDo()** – fires a custom callback method of your choice on every frame after a set timelimit has been reached.
- **OnEventTimerDoRepeat()** – fires a custom callback method of your choice after timelimit has been reached, and then resets the timer resulting in repeat actions at regular intervals.
- **OnEventTimerDoOnce** – same as OnEventTimerDo except it fires on one frame upon reaching the time limit.
- **UnParent()** – called when the OnStoppedFiring event is raised. It unparents the shot from the emitter if it was parented. Can be overriden for custom functionality.
- **OnOutBounds()** – called when a bullet reaches past the boundary limit set by the GlobalShotManager. Can be overriden for custom functionality.
- **RePoolOrDestroy()** – method that's called whenever a bullet is no longer needed in the scene. The shot

is repoooled if it inherits IRePoolable and repooling was enabled in the shot prefab, otherwise it's destroyed.
- **RePool()** – performs shot repooling. In some cases, minor customization to this procedure (reseting stats and so forth) are necessary so it can be overriden if required.
- **SetSprite()** – optional sprite frame replacement given from the Sprite Override in the firing script. Can be overriden if cutomizations are required.

## 8. Custom Shot Examples

The following are a few examples of how to easily create custom shot behaviors by making use of included events and custom method overrides. The examples covered below are included in the demo scenes called "TimeBulletExplode," "BulletsToCoins" and "ExpandingCircularBullet."

## 8.1. Bullets To Collectibles

This is a simple script which transforms bullets when the emitter that fired them is destroyed (typically when an enemy/player dies).

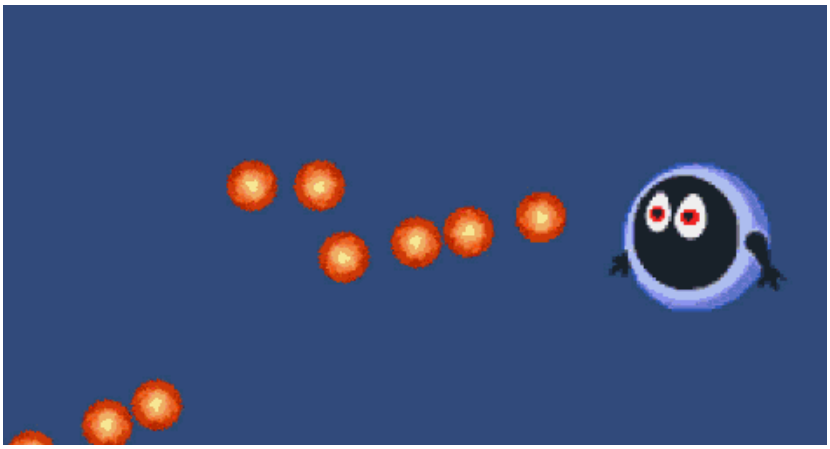
```
namespace ND VariaBULLET A
{
    public class CoinOnDestroyed : ShotLinearNonPhysics B
    {
        public GameObject Coin;

        public override void Update() C
        {
            base.Update();

        D OnEmitterDestroyedDoOnce(
                shot => {
                    GameObject coin = GameObject.Instantiate(Coin);
                    coin.transform.position = shot.transform.position;
                    coin.transform.rotation = shot.transform.rotation;

                    RePoolOrDestroy(); E
                }
            );
        }
    }
}
```

In this example we made sure to [A] reference the namespace that all shots belong to. [B] inherit from one of the existing shot classes. [C] Override the base Update() and implement our own procedure after all base Update() methods have been called. [D] Call the OnEmitterDestroyedOnce() event that fires once after the emitter has been destroyed, and pass in a custom procedure. [E] Call RePoolOrDestroy() directly to perform cleanup, destroying the unwanted bullets.



The resulting behavior instantiates the Coin prefab, producing the effect of turning the shots into collectible items when the enemy is destroyed.

## 8.2. Exploding Bullets

This script is a bit more complex than the last. The script is attached to a bullet which also has its own child controller/emitter nested under it. After a set time, the parent bullet is destroyed and triggers the child controller to produce a shot, creating the appearance of an exploding effect.

---

```
namespace ND VariaBULLET
{
    public class ShotExploding : ShotNonPhysics, IRePoolable A
    {
        [Range(1, 20)]
        public int ExplodeTimer = 1;

        [Range(1, 10)]
        public int SlowdownRate = 1;

        private BasePattern childController;
        private SpriteRenderer rend;

        public override void Start() B
        {
            base.Start();

            childController = transform.GetChild(0).GetChild(0).GetComponent<BasePattern>();
            rend = GetComponent<SpriteRenderer>();
        }

        public override void Update()
        {
            base.Update();

        C OnEventTimerDo(
                shot =>
                {
                    shot.ShotSpeed -= SlowdownRate * Time.deltaTime * shot.ShotSpeed / 2 * scale;

                    if (shot.ShotSpeed < 5 & shot.ShotSpeed > 2)
                    {
                        childController.TriggerAutoFire = true;
                        rend.enabled = false; //makes invisible immediately before destroying
                    }
                    else if (shot.ShotSpeed < 2)
                        RePoolOrDestroy(); E

                }, ExplodeTimer * 5
            );
        }

        public override void RePool(IPooler poolingScript) D
        {
            childController.TriggerAutoFire = false;
            rend.enabled = true;
            base.RePool(poolingScript);
        }
    }
}
```

You'll notice that this script [A] inherits from IRePoolable, which allows the parent bullet to be repooled. [B] Start() is overriden to set up connections to the renderer as well as the child controller. [C] An EventTimerDo event is used to set when the trigger for the child controller occurs. [D] The RePool routine called by [E]

RePoolOrDestroy() has custom procedures to reset the state of a couple of custom fields.

The resulting behavior produces a bullet which appears to explode after a set time.

## 8.3. Shots With Emitters

A similar variant of the ExplodingBullet script above, the following custom shot script uses an invisible shot which carries a child controller/emitter pattern that it fires. Since the child emitter firing script's **Parent To Emitter** setting is set to "Always", it creates a pattern that follows in the direction that the invisible parent bullet travels.

```
namespace ND VariaBULLET
{
    public class ShotEmitPattern : ShotNonPhysics
    {
        public bool shootOnce = true;
        private BasePattern childController;

        public override void Start()
        {
            base.Start();
            childController = transform.GetChild(0).GetChild(0).GetComponent<BasePattern>();
            childController.TriggerAutoFire = true;
        }

        public override void Update()
        {
            if (shootOnce)
            {
                OnEventTimerDoOnce(o => {
                    childController.TriggerAutoFire = false;
                }, 1);
            }

            base.Update();
        }

        protected override void setSprite(SpriteRenderer sr)
        {
            FireBullet fireScript = this.FiringScript as FireBullet;
            if (fireScript.SpriteOverride == null)
                return;

            Transform child = transform.GetChild(0).GetChild(0);

            foreach (Transform emitter in child)
            {
                foreach (Transform point in emitter)
                {
                    FireBullet pointScript = point.GetComponent<FireBullet>();
                    pointScript.SpriteOverride = fireScript.SpriteOverride;
                }
            }
        }
    }
}
```

Note that setSprite() is overriden to produce custom behavior (in this case, passing the sprite override from the parent firing script to the child if it is set.

---

The emitter pattern set as a child to the invisible shot is a simple radial with a slow Shot Speed, resulting in an expanding circular shot.