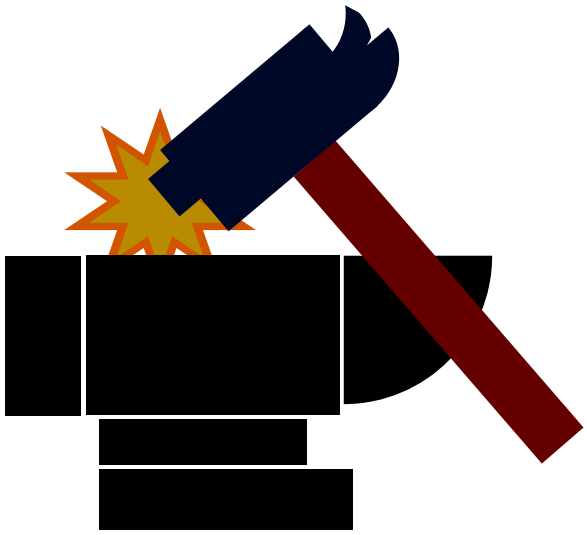

GunForger

Version 1.0.0

Rex Gong

December 12, 2025

Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

1.1 System Description

The **GunForger** is a comprehensive, modular 2D weapon framework for Unity game development. It provides a complete solution for implementing ranged weapons in 2D games, supporting multiple genres including side-scrollers, top-down shooters, and twin-stick arena games.

1.2 Problem Space

Modern 2D games require flexible, reusable weapon systems that can adapt to various gameplay styles. Traditional approaches often involve:

- Hard-coded weapon behaviors that are difficult to modify
- Lack of visual feedback systems (recoil, spread, effects)
- No built-in tools for rapid prototyping and balancing
- Complex integration with game systems (UI, audio, animation)

The GunForger system addresses these challenges through a modular, data-driven architecture that separates configuration from logic, provides extensive visual feedback, and includes editor tools for rapid development. The system draws inspiration from established 2D game development patterns [4] and incorporates best practices from community tutorials [2, 3].

1.3 Key Features

- **Modular Architecture**: Separates weapons, projectiles, and health systems
- **Multiple Game Styles**: Supports side-scroller [2], top-down [3], and full-rotation aiming
- **Weapon Presets**: ScriptableObject-based configuration system
- **Spread System**: Time-based, shot-based, or hybrid accuracy models
- **Visual Effects**: Recoil, muzzle flashes, impact effects, AOE damage
- **Editor Integration**: Custom inspectors, prefab generators, and utility tools

- **Health System**: Comprehensive damage, healing, and death management
- **Extensible Events**: UnityEvents for easy game logic integration

2 Installation and Usage

2.1 Prerequisites

Before installing the GunForger, ensure your development environment meets the following requirements:

Requirement	Version/Details
Unity Editor	2022.3 LTS or newer
Unity 2D Package	Built-in (no additional install)
Physics 2D Package	Built-in (no additional install)

Table 1: System prerequisites and requirements

Recommended Unity Packages:

- TextMeshPro - For UI such as Ammo Count and Weapon Names (Is Included in Demo)

2.2 File Structure

The system consists of the following key components:

File	Purpose
`Runtime/RangedWeapon.cs`	Main weapon controller
`Runtime/Projectile.cs`	Projectile behavior and AOE
`Runtime/Health.cs`	Health and damage system
`Runtime/WeaponPreset.cs`	ScriptableObject configuration
`Editor/WeaponEditor.cs`	Custom editor tools
`Editor/WeaponPresetGenerator.cs`	Prefab generation utilities

Table 2: Core system files

2.3 Quick Start Guide

2.3.1 Step 1: Create a Basic Weapon

1. In Unity, navigate to: [`Assets → 2D Weapon System → Weapon Preset`] or [Assets → 2D Weapon System → Weapon Prefabs] for pre-made weapons
2. A new weapon GameObject will be created with all required components
3. Assign a projectile prefab to the `Projectile Prefab` field

2.3.2 Step 2: Configure Weapon Properties

Configure your weapon using the custom inspector in Ranged Weapon.cs:

- Set `Game Type` (Side Scroller, Top Down, or Full Rotation)
- Adjust fire rate, damage, spread, and other properties
- Use the `Quick Stats` section for balancing feedback

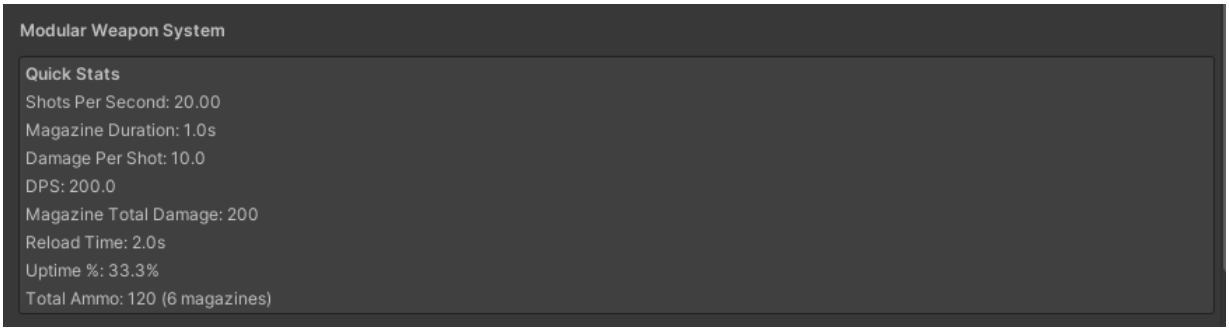

Figure 1: Weapon statistics in Unity Inspector under RangedWeapon.cs

2.3.3 Step 3: Create Projectile Prefab

Click the `Setup Basic Projectile` button to generate a complete projectile prefab with all required components. The projectile system uses Unity's 2D physics engine [5] for collision detection.

3 Samples and Demonstrations

3.1 Sample Package Structure

The GunForger Weapons System includes a comprehensive sample package with ready-to-use assets, scenes, and prefabs. After importing the package, you can access the samples through Unity's Package Manager:

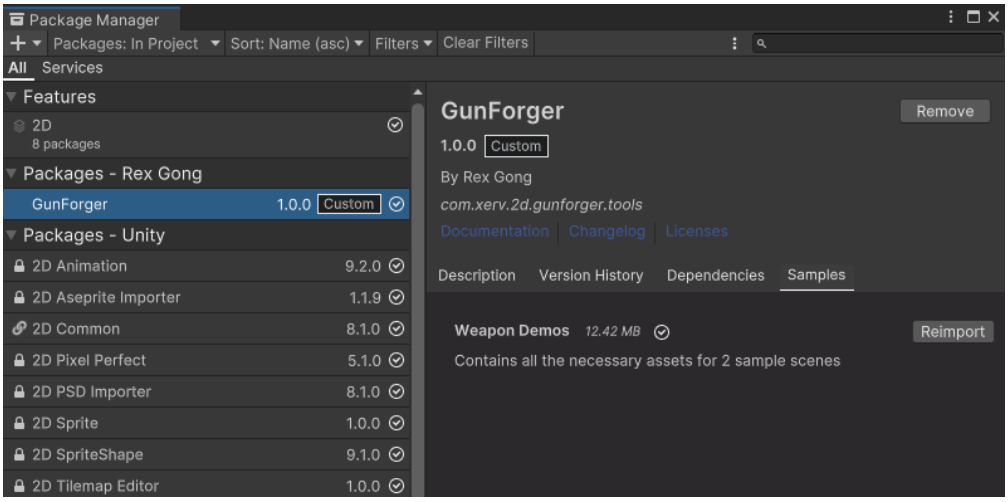

Figure 2: Accessing GunForger samples through the Package Manager

3.2 Sample Contents

The sample package contains the following organized folders:

Folder	Contents
`Animations`	Character and weapon animation clips, controllers, and blend trees for various movement and firing states
`Art`	Sprite sheets, UI elements, and visual assets including weapon sprites, character artwork, and environment assets
`Audio`	Sound effects for weapons (fire, reload, empty clip), impacts
`Guns`	Pre-configured weapon prefabs including pistol, assault rifle, shotgun, sniper rifle, etc
`Physical Materials`	Just Player Slip for Side-Scroller 2D games
`Player`	Complete player character prefabs with input handling, movement, and weapon switching systems
`Scenes`	Demonstration scenes for different game genres: • `SideScrollerDemo.unity`: Platformer-style shooting mechanics • `TopDownDemo.unity`: Twin-stick shooter implementation
`Scripts`	Additional demonstration scripts including: • Weapon selection Script • Ammo management UI • Camera controllers for different perspectives

Table 3: Sample package contents and descriptions

3.3 Importing and Using Samples

3.3.1 Import via Package Manager

1. Open Unity's Package Manager (`Window → Package Manager`)
2. Select the GunForger package from the list
3. Click on the **Samples** tab in the package details pane
4. Click the **Import** button next to "GunForger Demo Assets"
5. Unity will import all sample assets into your project at: `Assets/Samples/GunForger/[vers`

4 Projectile System and Collision Detection

4.1 Dual Detection System Architecture

The GunForger projectile system employs a sophisticated dual-detection approach to ensure reliable collision detection across all gameplay scenarios. This design addresses the fundamental challenge in fast-paced 2D games: preventing fast-moving projectiles from tunneling through targets.

4.1.1 Raycast-Based Detection for High-Velocity Projectiles

For projectiles moving at high speeds (common with sniper rifles and railguns), the system implements continuous collision detection through raycasting between positions:

$$\vec{d} = \vec{p}_{\text{current}} - \vec{p}_{\text{previous}}$$

$$\text{Raycast: Physics2D.Raycast}(\vec{p}_{\text{previous}}, \hat{d}, \|\vec{d}\|, \text{hitLayers})$$

```
// Implementation of raycast-based collision detection
void CheckRaycastHit()
{
    Vector2 currentPosition = transform.position;
    Vector2 direction = currentPosition - lastPosition;
    float distance = direction.magnitude;

    if (distance > 0)
    {
        // Cast ray from last position to current position
        RaycastHit2D hit = Physics2D.Raycast(
            lastPosition,
            direction.normalized,
            distance,
            hitLayers
        );

        if (hit.collider != null && !hasHit)
        {
            // Prevent self-hits by checking owner tag
```

```
            if (!string.IsNullOrEmpty(ownerTag) &&
                hit.collider.CompareTag(ownerTag))
                return;

            hasHit = true;
            ProcessHit(hit.collider, hit.point);
        }
    }
    lastPosition = currentPosition;
}
```

4.1.2 Trigger-Based Fallback Detection

For slower projectiles and edge cases, the system uses Unity's trigger collision system as a reliable fallback:

```
void OnTriggerEnter2D(Collider2D other)
{
    if (hasHit) return; // Already processed a hit

    // Layer filtering using bitmask operations
    if (((1 << other.gameObject.layer) & hitLayers) == 0)
        return;

    // Owner collision prevention
    if (!string.IsNullOrEmpty(ownerTag) &&
        other.CompareTag(ownerTag))
        return;

    hasHit = true;
    Vector2 impactPoint = other.ClosestPoint(transform.position);
    ProcessHit(other, impactPoint);
}
```

4.2 Area-of-Effect (AOE) Damage Implementation

The system provides comprehensive AOE damage with configurable parameters and performance optimizations.

4.2.1 Mathematical Model for AOE Damage

AOE damage follows an inverse linear falloff model:

$$D(r) = D_{\max} \times \left(1 - \frac{r}{R}\right) \quad \text{for } 0 \leq r \leq R$$

Algorithm 1 AOE Damage Application Algorithm

```
procedure ApplyAOEDamage(center, directHitTarget)
    hits ← Physics2D.OverlapCircleAll(center, aoeRadius, hitLayers)
    for hit ∈ hits do
        if hit.gameObject = directHitTarget then
            continue                                  ▷ Skip directly hit target
        end if
        if !aoeDamagesFriendlies and hit.CompareTag("Player") then
            continue                                  ▷ Respect friendly fire setting
        end if
        health ← hit.GetComponent < Health > ()
        if health ≠ null then
            distance ← ||center − hit.transform.position||
            damageMultiplier ← 1 − distance / aoeRadius
            damage ← aoeDamage × damageMultiplier
            health.TakeDamage(damage)
        end if
    end for
end procedure
```

4.2.2 Performance Optimizations

- **Layer Mask Filtering**: Only checks objects in specified hitLayers
- **Squared Distance Checks**: Avoids expensive square root operations until necessary
- **Early Exit Conditions**: Skips friendly targets and direct hit objects immediately

4.3 Owner Tagging and Collision Filtering

To prevent self-inflicted damage and team kills, the system implements a tagging system:

```
        // Ignore owner collisions
        if (!string.IsNullOrEmpty(ownerTag) && other.CompareTag(ownerTag))
            return;

        // Check if the other object's layer is among hitLayers
        if (((1 << other.gameObject.layer) & hitLayers) == 0)
        {
            Debug.Log($"Ignoring collision with {other.gameObject.name} on
             ↪  layer {LayerMask.LayerToName(other.gameObject.layer)}");
            return;
        }
        else if (other.CompareTag("Player") && !aoeDamagesFriendlies)
        {
            // Respect friendly-fire toggle for players

```

This system allows for:

- **Self-damage prevention**: Projectiles ignore their creator
- **Team-based damage**: Configurable friendly fire settings
- **Multiplayer support**: Each projectile tracks its origin

5 Health System Integration

5.1 Component Architecture and Event System

The Health component provides a complete damage and life management system with extensive visual feedback and event hooks for game logic integration.

5.2 Damage Processing Pipeline

The Health component processes damage through a comprehensive pipeline:

```
public void TakeDamage(float damage)
{
    // 1. State validation
    if (isInvulnerable || isDead) return;

    // 2. Health modification
    currentHealth -= damage;
    currentHealth = Mathf.Max(currentHealth, 0);

    // 3. Event invocation
    OnDamageTaken?.Invoke(damage);
    OnHealthChanged?.Invoke(currentHealth);

    // 4. Visual feedback
    if (damageEffectPrefab != null)
        Instantiate(damageEffectPrefab, transform.position,
        ↪  Quaternion.identity);

    // 5. Sprite flashing
    if (spriteRenderer != null)
        StartCoroutine(DamageFlash());

    // 6. Death check
    if (currentHealth <= 0) Die();
}
```

5.3 Visual Feedback System

5.3.1 Damage Flash Effect

The system provides immediate visual feedback through sprite color flashing:

```
private System.Collections.IEnumerator DamageFlash()
{
    // Flash to damage color
    spriteRenderer.color = damageFlashColor;

    // Wait for specified duration
    yield return new WaitForSeconds(flashDuration);

    // Restore original color
    spriteRenderer.color = originalColor;
}
```

5.3.2 Configurable Visual Parameters

Parameter	Default	Description
`damageFlashColor`	Red	Color to flash when taking damage
`flashDuration`	0.1s	Duration of damage flash effect
`damageEffectPrefab`	None	Optional particle effect on damage
`deathEffectPrefab`	None	Particle effect on death
`deathDelay`	0s	Delay before object destruction

Table 4: Configurable visual feedback parameters in Health component

5.4 Event-Driven Architecture

The Health component exposes UnityEvents for seamless integration with other game systems:

```
public UnityEvent<float> OnDamageTaken;
```

```
public UnityEvent<float> OnHealthChanged;
public UnityEvent OnDeath;

private Color originalColor; // Cached original sprite color for flashing
private bool isDead = false; // Tracks death state to prevent re-entry

void Awake()
{
    // Ensure starting health is initialized
    currentHealth = maxHealth;

    // Cache sprite renderer if not assigned and store original color
    if (spriteRenderer == null)
        spriteRenderer = GetComponent<SpriteRenderer>();

    if (spriteRenderer != null)
        originalColor = spriteRenderer.color;
}
public void TakeDamage(float damage)
{
    if (isInvulnerable || isDead)
        return;

    currentHealth -= damage;
    currentHealth = Mathf.Max(currentHealth, 0);

    // Notify listeners
    OnDamageTaken?.Invoke(damage);
    OnHealthChanged?.Invoke(currentHealth);

    // Optional damage VFX
    if (damageEffectPrefab != null)
    {
        Instantiate(damageEffectPrefab, transform.position,
        ↪  Quaternion.identity);
    }

    // Sprite flash visual feedback
    if (spriteRenderer != null)
    {
        StopAllCoroutines();
```

```
        StartCoroutine(DamageFlash());
    }

    if (currentHealth <= 0)
    {
        Die();
    }
```

5.5 Death Processing and Cleanup

The death handling system ensures proper cleanup and effect spawning:

Algorithm 2 Death Processing Algorithm

procedure DIE
  if *isDead* **then return**
  end if ▷ Prevent re-entry
  isDead ← *true*
  OnDeath.Invoke() ▷ Notify listeners
  if *deathEffectPrefab* ≠ *null* **then**
    Instantiate(*deathEffectPrefab*, *transform.position*, *Quaternion.identity*)
  end if
  if *destroyOnDeath* **then**
    Destroy(*gameObject*, *deathDelay*) ▷ Optional delay
  end if
end procedure

5.6 Invulnerability States

The system supports temporary invulnerability for gameplay mechanics like power-ups or dodging:

```
public void TakeDamage(float damage)
//.......code.......
    if (isInvulnerable || isDead)
        return;
//......code........
```

```
    public void SetInvulnerable(bool invulnerable) => isInvulnerable =
    ↪  invulnerable;
```

6 Editor Tools and Development Workflow

6.1 Custom Inspector System

The GunForger system includes comprehensive custom editors in the Unity Inspector, giving you a powerful weapon design tool.

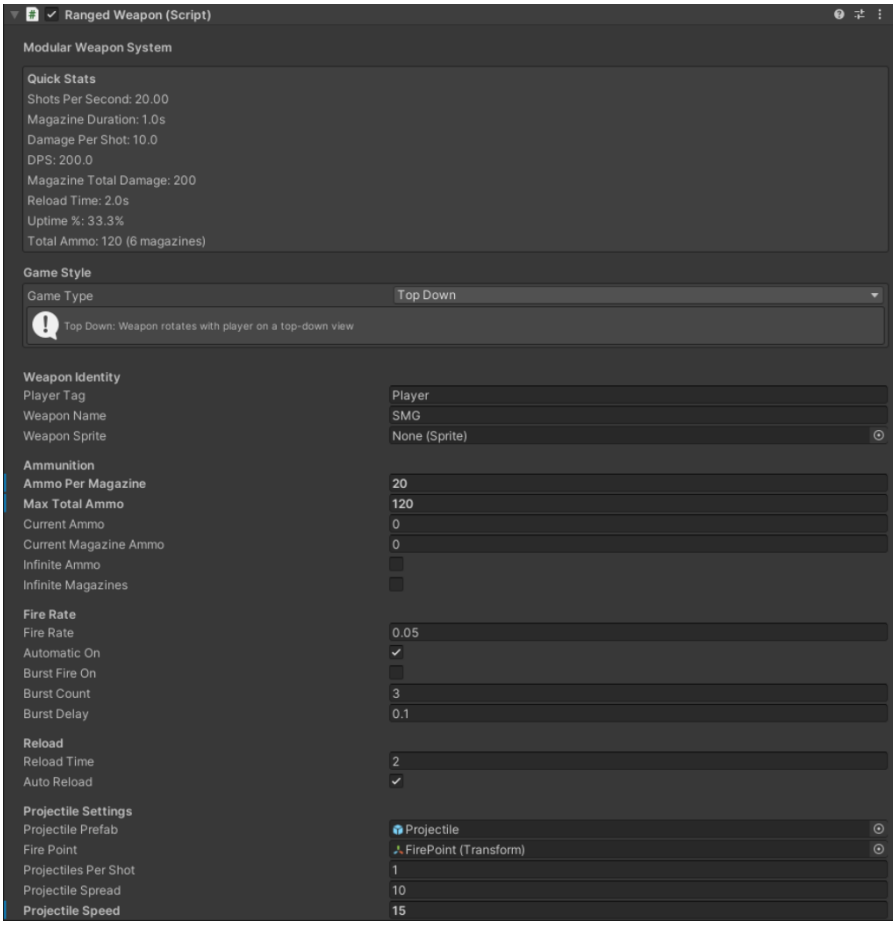

Figure 3: Custom Weapon Modification Menu in the Unity Inspector for Assets with the Script

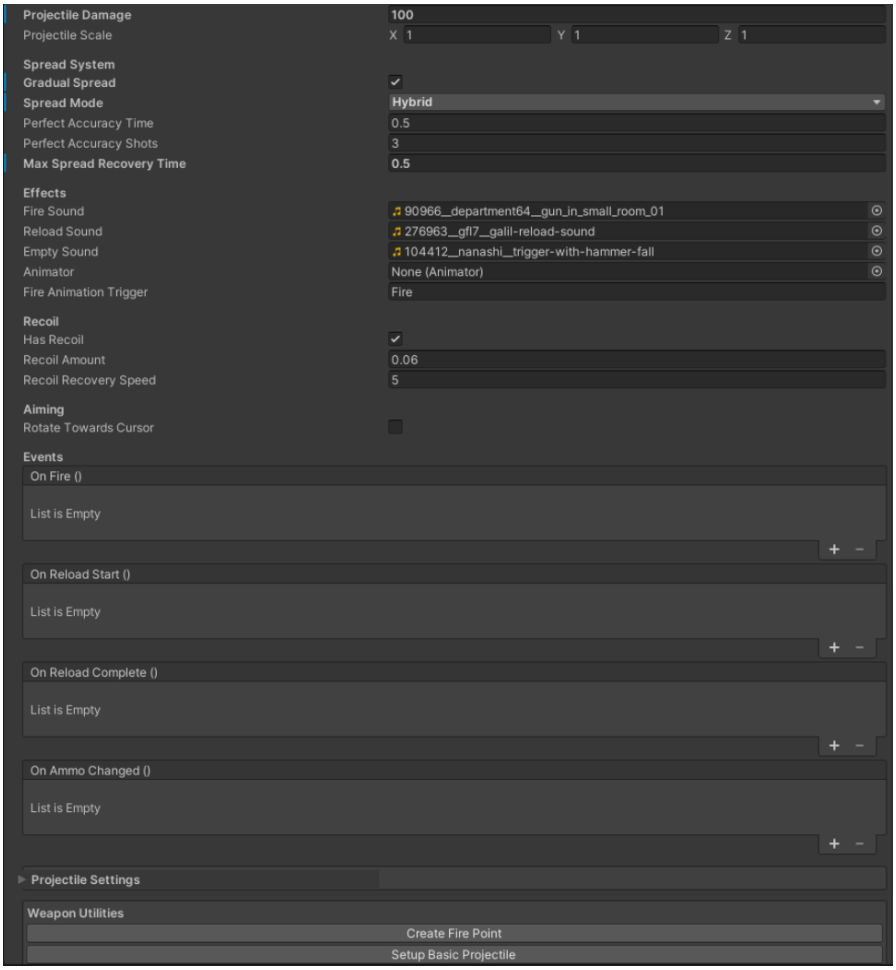

Figure 4: Custom Weapon Modification Menu in the Unity Inspector for Assets with the Script (Continued)

6.1.1 Inline Projectile Editing

A unique feature allows designers to edit projectile properties without leaving the weapon inspector:

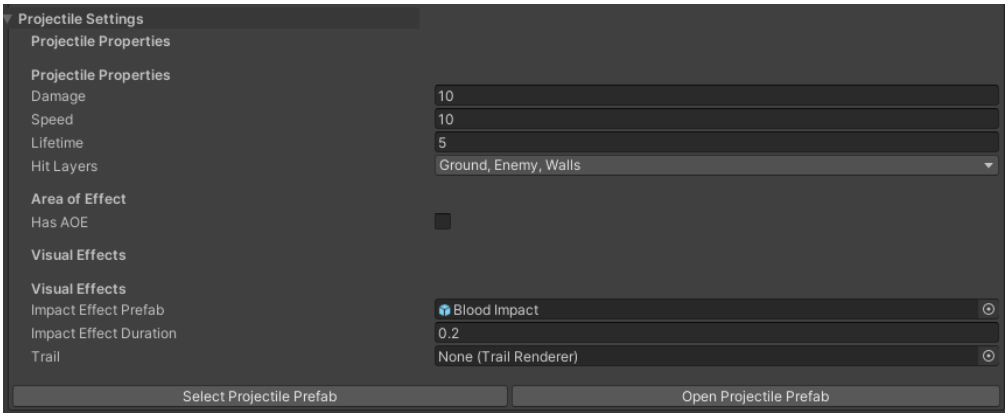

Figure 5: Weapon Menu includes projectile editing for the projectile prefab directly

6.2 Weapon Preset Generator

The system includes a powerful prefab generator that creates 10 pre-configured weapon archetypes:

Weapon	Key Characteristics
Pistol	Semi-auto, 12 mag, 15 dmg, 0.15s rate
Assault Rifle	Full-auto, 30 mag, 12 dmg, 5° spread
Shotgun	8 pellets, 25° spread, 8 dmg each
Sniper Rifle	100 dmg, perfect accuracy, 1.5s rate
SMG	0.05s rate, 40 mag, 10° spread
Burst Rifle	3-round burst, 15 dmg, 3° spread
Rocket Launcher	50 dmg, 1.5m AOE, slow projectile
Minigun	0.03s rate, 200 mag, 15° spread
Plasma Rifle	Infinite ammo, 20 dmg, 0.2s rate
Flamethrower	0.02s rate, 3 dmg, area denial

Table 5: Weapon preset generator configurations (compact view)

6.2.1 Reflection-Based Configuration

The generator uses reflection to set private fields, allowing complete weapon configuration:

```
private static void SetField(RangedWeapon weapon, string fieldName,
    object value)
{
    var field = typeof(RangedWeapon).GetField(fieldName,
        System.Reflection.BindingFlags.NonPublic |
        System.Reflection.BindingFlags.Public |
        System.Reflection.BindingFlags.Instance);

    if (field != null)
    {
```

```
            field.SetValue(weapon, value);
        }
        else
        {
            Debug.LogWarning($"Field '{fieldName}' not found on
            ↪ RangedWeapon");
        }
    }
```

6.3 Scene Creation Utilities

Quick creation tools for rapid prototyping:

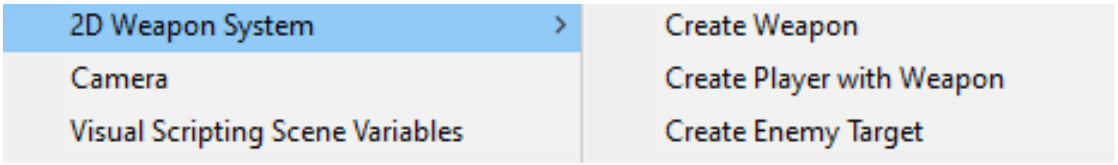

Figure 6: Can be found in hierarchy

6.4 Development Workflow

1. **Rapid Prototyping**: Use menu items to create player, enemy, and weapons
2. **Weapon Creation**: Generate pre-configured weapons or create custom ones
3. **Projectile Setup**: Use "Setup Basic Projectile" button for quick projectile creation
4. **Balancing**: Use Quick Stats section for DPS and uptime calculations
5. **Testing**: Enable Gizmos to visualize spread, fire points, and AOE radii
6. **Preset Creation**: Save configurations as ScriptableObject presets for reuse

6.5 Debugging and Visualization Tools

- **Gizmos**: Visualize fire points, spread cones, and AOE radii in Scene view
- **Console Logging**: Detailed hit detection and damage calculation logs

- **Runtime Information**: Projectile speed and damage displayed in inspector during play mode
- **Custom Handles**: Editor-only visualization for complex weapon behaviors

7 Advanced Weapon Features and Implementation

7.1 Burst Fire System

The burst fire implementation provides controlled automatic fire with precise timing control.

Algorithm 3 Burst Fire Coroutine Algorithm

```
1:  procedure FireBurst
2:      isFiring ← true
3:      shotsFired ← 0
4:      while shotsFired < burstCount and currentMagazineAmmo > 0 do
5:          Fire()                                          ▷ Single shot logic
6:          shotsFired ← shotsFired + 1
7:          if shotsFired < burstCount then
8:              WaitForSeconds(burstDelay)
9:          end if
10:     end while
11:     isFiring ← false
12:     nextFireTime ← Time.time + fireRate                 ▷ Enforce cooldown
13: end procedure
```

7.2 Spread System Implementation

The gradual spread system implements four distinct accuracy models with configurable parameters.

7.2.1 Spread Multiplier State Tracking

Variable	Type	Purpose
`lastShotTime`	float	Timestamp of last shot for time-based spread calculations
`consecutiveShots`	int	Count of consecutive shots fired without recovery period
`currentSpreadMultiplier`	float	Current accuracy state (0 = perfect accuracy, 1 = maximum spread)
`perfectAccuracyTime`	float	Time window (seconds) for perfect accuracy recovery
`perfectAccuracyShots`	int	Number of initial shots with perfect accuracy before spread increases

Table 6: Spread system state variables and their purposes

7.2.2 Hybrid Spread Model Implementation

The hybrid model combines time-based and shot-based accuracy for realistic weapon behavior:

```
1  private void CalculateSpreadMultiplier()
2  {
3      float timeSinceLastShot = Time.time - lastShotTime;
4
5      switch (spreadMode)
6      {
7          case SpreadMode.Hybrid:
8              // Reset if enough time has passed
9              if (timeSinceLastShot > maxSpreadRecoveryTime)
10             {
11                 consecutiveShots = 0;
12                 currentSpreadMultiplier = 0f;
13             }
14             else
15             {
```

```
                consecutiveShots++;

                // Perfect accuracy for first N shots
                if (consecutiveShots <= perfectAccuracyShots)
                {
                    currentSpreadMultiplier = 0f;
                }
                else
                {
                    // Combine time and shot factors
                    float shotFactor = Mathf.Clamp01(
                        (float)(consecutiveShots - perfectAccuracyShots) /
                        perfectAccuracyShots
                    );
                    float timeFactor = Mathf.Clamp01(
                        1f - (timeSinceLastShot / perfectAccuracyTime)
                    );

                    // Use the more restrictive factor
                    currentSpreadMultiplier = Mathf.Max(shotFactor,
                    ↪  timeFactor);
                }
            }
            break;

        // Other spread modes...
    }

    lastShotTime = Time.time;
}
```

7.3 Multiple Projectile Distribution

For shotgun-style weapons, the system provides even projectile distribution across the spread angle:

$$\theta_i = -\theta_{\text{max}} + \left(\frac{2\theta_{\text{max}}}{n-1}\right) \times i$$

where:

- θ_i: Angle of projectile i (degrees)
- θ_{max}: Maximum spread angle
- n: Number of projectiles per shot
- i: Projectile index (0 to $n - 1$)

```
void SpawnProjectile(int index)
{
    //.........code..........
    float spreadAngle = 0f;

    if (projectilesPerShot > 1)
    {
        // Even distribution across spread range
        float totalSpread = projectileSpread * 2;
        spreadAngle = -projectileSpread +
                    (totalSpread / (projectilesPerShot - 1)) * index;
    }

    Quaternion spreadRotation = Quaternion.Euler(0, 0, spreadAngle);
    Vector2 direction = spreadRotation * firePoint.right;

     //.........code..........
}
```

7.4 Recoil Implementation Details

The recoil system provides physical feedback through weapon displacement with smooth recovery.

7.4.1 Recoil Direction Calculation

```
void ApplyRecoil()
{
    if (SideScroller || FullRotation)
    {
        // Side-scroller: recoil opposite to fire direction
        if (Player != null && Player.transform.localScale.x < 0)
```

```
            recoilOffset = firePoint.right * recoilAmount;
        else
            recoilOffset = -firePoint.right * recoilAmount;
    }
    else
    {
        // General case: recoil opposite to aim direction
        Vector2 mouseWorldPos =
        ↪ Camera.main.ScreenToWorldPoint(Input.mousePosition);
        Vector2 shootDirection = (mouseWorldPos -
        ↪ (Vector2)transform.position).normalized;
        recoilOffset =
        ↪ transform.InverseTransformDirection(-shootDirection) *
        ↪ recoilAmount;
    }
}
```

7.4.2 Exponential Recoil Recovery

Recoil offset decays exponentially for smooth visual feedback:

$$\vec{r}(t) = \vec{r}_0 \times e^{-\alpha t}$$

where α is the recoil recovery speed.

```
    void HandleRecoilRecovery()
    {
        if (recoilOffset.magnitude > 0.001f)
        {
            // Smoothly interpolate offset back to zero using recovery speed
            recoilOffset = Vector3.Lerp(recoilOffset, Vector3.zero,
            ↪ Time.deltaTime * recoilRecoverySpeed);
            transform.localPosition = originalPosition + recoilOffset;
        }
        else
        {
            transform.localPosition = originalPosition;
        }
    }
```

7.5 Performance Optimization Techniques

The system implements several optimizations for efficient runtime performance:

Optimization	Implementation
Early Exit Conditions	Multiple guard clauses prevent unnecessary calculations
Layer Mask Filtering	Physics queries restricted to relevant layers only
Object Pooling Ready	Modular architecture supports easy object pooling integration
Event Consolidation	Events batched and invoked at appropriate times

Table 7: Performance optimization techniques implemented in the system

8 System Applications

8.1 Game Genre Support

The GunForger system supports multiple 2D game genres by adapting its aiming and rotation logic. Table 8 shows how to configure the system for each genre, incorporating techniques from popular tutorials [2, 3].

Genre	Configuration and Reference
Side-Scroller	Set `SideScroller = true`. The weapon sprite flips horizontally based on aiming direction, suitable for platformers and metroidvania games [2].
Top-Down	Set `TopDown = true`. The weapon rotates freely around the player character, ideal for twin-stick shooters and RPG combat [3].
Full Rotation	Set `FullRotation = true`. Enables 360-degree aiming without sprite flipping, perfect for arena shooters.

Table 8: Genre-specific configurations for the GunForger system

8.2 Performance Considerations

The system implements several optimizations based on Unity best practices [5] to ensure smooth performance even in demanding scenarios:

- **Object Pooling**: For high-volume weapon scenarios, implement object pooling for projectiles and visual effects. The system's modular architecture makes this straightforward to implement.
- **Raycast Optimization**: The projectile system uses targeted raycasts with specific layer masks and distance limits to minimize performance impact during collision detection.
- **Event Management**: While UnityEvents provide excellent decoupling, they introduce overhead. The system uses them judiciously, avoiding events in performance-critical update loops.

- **Editor Code Isolation**: All editor utilities and custom inspectors are conditionally compiled with `#if UNITY_EDITOR` directives, ensuring they don't affect runtime performance in builds.
- **Efficient Physics**: The projectile system leverages Unity's 2D physics engine [5] with appropriate collision detection modes (Continuous for fast projectiles) to balance accuracy and performance.

8.3 Integration with Game Systems

The GunForger system is designed for seamless integration with common game systems:

System	Integration Approach
UI/HUD	Subscribe to `OnAmmoChanged` and `OnHealthChanged` events for real-time updates
Audio System	Weapon sounds are managed through Unity's AudioSource with configurable clips
Animation	Use animator triggers (`Fire`, `IsReloading`) for character and weapon animations
Save/Load	Serialize weapon configurations and ammo counts for game state persistence
AI System	AI controllers can access weapon properties and call `Fire()` method directly

Table 9: Integration points with common game systems

9 Troubleshooting and Common Issues

9.1 Frequent Problems and Solutions

Issue	Solution and Reference
Weapon doesn't fire	Check projectile prefab assignment and hit layers [3]
No sound effects	Ensure AudioSource component exists and clips are assigned
Weapon doesn't rotate	Verify rotateTowardsCursor is enabled and camera is tagged Main [2]
Projectiles go through enemies	Enable Continuous collision detection on Rigidbody2D [5]
Editor tools not appearing	Ensure scripts are in Editor folder and UNITY_EDITOR is defined
Spread not working	Check gradualSpread is enabled and spreadMode is not None [1]

Table 10: Common troubleshooting guide

9.2 Debugging Tools

The system includes built-in debugging visualizations based on Unity development practices:

- **Gizmos**: Show fire point, spread cone, and AOE radius in Scene view
- **Console Logs**: Detailed logging for hits, AOE calculations, and errors
- **Editor Stats**: Real-time DPS and balancing information

9.3 Troubleshooting Sample Issues

Issue	Solution
Samples not appearing in Package Manager	Ensure you have the correct version of the package installed. Try refreshing the Package Manager window.
Missing dependencies after import	The samples require the core GunForger system. Ensure the main package is properly installed first.
Input not working in demo scenes	Verify that Unity's Input System package is installed and enabled in Project Settings.
Weapons not firing in samples	Check that the player prefab has all required components and that projectile prefabs are properly assigned.

Table 11: Common sample-related issues and solutions

10 Conclusion

10.1 Summary

The GunForger provides a complete, modular solution for 2D weapon implementation in Unity. Its key strengths are:

- **Flexibility**: Supports multiple game genres and weapon types
- **Developer Experience**: Comprehensive editor tools and visual feedback
- **Performance**: Optimized for 2D physics and rendering [5]
- **Extensibility**: Clean architecture for customization and extension

10.2 Academic and Technical Foundation

The system builds upon established game development principles [4] and incorporates techniques from popular Unity tutorials [1–3]. The mathematical models for spread, AOE, and recoil are grounded in practical game physics implementations.

References

[1] Unity Assets. Advanced weapon system for unity 2d, 2020.

[2] Blackthornprod. Making a sidescroller shooting system in unity, 2019.

[3] Brackeys. How to make a 2d top-down shooter in unity, 2018.

[4] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[5] Unity Technologies. Unity 2d physics engine documentation. 2023.