Tower Defence Notes : Turret System

23 June 2026 11:02

Turret System

Overview

The Turret System is responsible for detecting enemies, selecting targets, rotating towards them, and launching projectiles. The goal of this implementation was to create a reusable turret framework that could support multiple turret types without requiring changes to the core combat logic.

Instead of hardcoding values directly inside the turret controller, gameplay statistics such as damage, range, fire rate, projectile type, and special effects are stored separately using Scriptable Objects. This allows new turret variations to be created through configuration rather than additional code.

The system is built around a simple combat pipeline:

Target Detection → Target Selection → Rotation → Projectile Launch → Damage Delivery

Key Features

- Data-driven turret configuration
- Intelligent target acquisition
- Projectile-based combat
- Smooth turret rotation
- Object pooling for projectiles
- Event-driven turret registration
- Support for future special attacks

Architecture Overview

The turret system is divided into two main components.

TurrentStats

Stores gameplay data such as:

- Damage
- Fire Rate
- Attack Range
- Projectile Type
- Upgrade Cost
- Health
- Special Effects

TurrentController

Handles runtime behaviour including:

- Target detection
- Target selection
- Rotation
- Shooting
- Projectile management
- Communication with other systems

Turret Configuration

All turret statistics are stored inside a Scriptable Object.

```
[CreateAssetMenu]
public class TurrentStats : ScriptableObject
{
    public int damage;
    public float firerate;
    public float totalRange;
    public float dangerZoneRange;
    public GameObject projectilePrefab;
}
```

When the turret is initialized, these values are loaded into runtime variables.

```
private void onIntitialiseValues()
{
    damage = turrentStats.damage;
    fireRate = turrentStats.firerate;
    totalRange = turrentStats.totalRange;
    dangerZone = turrentStats.dangerZoneRange;
}
```

This approach allows different turret types to share the same combat system while maintaining unique gameplay characteristics.

For example, a sniper turret, cannon turret, and freeze turret can all use the same controller while behaving differently through configuration.

Target Detection & Selection

The first responsibility of the turret is finding a valid target.

The system continuously scans for enemies within its attack radius.

```
Collider[] enemies =
Physics.OverlapSphere(
    canonTransform.position,
    totalRange,
    enemyLayer);
```

If no enemies are found, the turret remains idle.

When enemies are detected, the turret searches for targets that have entered its danger zone.

```
if(distance < dangerZone)
{
    foundTarget = enemy.transform;
    break;
}
```

The danger zone acts as a priority area, allowing nearby enemies to be targeted immediately.

If no suitable target is found, the turret can request assistance from external systems.

```
private void RequestEnemyTarget()
{
    onRequestEnemy?.Invoke(id);
}
```

This design allows future targeting systems to assign enemies based on additional criteria such as threat level, health, or path progression.

Rotation System

Once a target has been selected, the turret rotates toward it before firing.

```
Vector3 direction =
```

```
currentTarget.position -
canonTransform.position;
direction.y = 0;
```
The desired rotation is calculated using the direction to the target.
```
Quaternion targetRotation =
Quaternion.LookRotation(direction);
```
Instead of snapping instantly, the turret smoothly rotates using interpolation.
```
canonTransform.rotation =
Quaternion.Slerp(
   canonTransform.rotation,
   targetRotation,
   rotationSpeed * Time.deltaTime);
```
This creates more natural movement and improves visual feedback during combat.

Projectile-Based Combat

Rather than applying damage directly, the turret attacks by spawning projectiles.
The firing process is controlled through a timer.
```
if(fireTimer >= 1f / fireRate)
{
   Shoot(turrentStats.projectileIndex);
   fireTimer = 0f;
}
```
Using fire rate in this way ensures that attack speed remains configurable and independent of frame rate.
When a shot is fired, the projectile receives the current target and turret configuration.
```
projScript.SetTarget(
   currentTarget,
   turrentStats);
```
This keeps projectile behaviour independent from turret behaviour.
The turret focuses on firing, while the projectile is responsible for travelling and delivering damage.

Projectile Pooling

Creating and destroying projectiles repeatedly can become expensive as enemy counts increase.
To avoid unnecessary allocations, the system uses object pooling.
```
private Dictionary<int,
Queue<GameObject>>
projectilePrefabPool =
new Dictionary<int,
Queue<GameObject>>();
```
When a projectile is needed:

- Reuse an inactive projectile if available.
- Create a new projectile only when necessary.

```
if(projectilePrefabPool.ContainsKey(projectileIndex)
&& projectilePrefabPool[projectileIndex].Count > 0)
{
   prefab =
   projectilePrefabPool[projectileIndex]
   .Dequeue();
}
```
After its lifetime expires, the projectile is returned to the pool.
```
projectilePrefabPool
```

[projectileIndex]
.Enqueue(prefab);
This reduces memory allocations and helps maintain stable performance during large battles.

Turret Registration

Every turret registers itself when created.

```
private void RegisterOnSpawn()
{
   onSpawnTurrent?.Invoke(
      totalTurrentCount,
      this);
totalTurrentCount++;
}
```

The registration event allows other systems to become aware of newly placed turrets without requiring direct references.
Similarly, turrets unregister themselves when destroyed.

```
OnUnregisterTurrent?.Invoke(id);
```

This keeps the turret ecosystem synchronized throughout gameplay.

Design Decisions

Data-Driven Configuration

Turret statistics are separated from turret behaviour, making balancing easier and reducing duplicated code.

Projectile-Based Attacks

Projectiles create more flexibility than direct damage because they support future mechanics such as splash damage, homing missiles, and status effects.

Object Pooling

Pooling reduces runtime allocations and improves performance during high-intensity combat.

Event-Based Communication

Registration and targeting requests are handled through events, allowing systems to communicate without becoming tightly coupled.

Conclusion

The Turret System provides a flexible foundation for defensive gameplay. By separating configuration, targeting, combat, projectile management, and communication responsibilities, the system remains easy to maintain and extend.
The result is a reusable turret framework capable of supporting multiple turret types while maintaining consistent combat behaviour and good runtime performance.