RocketMAN

Michael Bui 293886
GMD-1 Course Project

Concept

RocketMAN is a casual first-person movement-focused racing game where the primary goal is getting through all maps using the least amount of rockets in the fastest time possible. The game is built around the two most satisfying movement mechanics (*in my opinion*) - Rocket Jumping and free range Grappling-Hook based swinging.

Arhitecture

The overall arhictecture is inspired by Vertical Slice Architecture, where the game functionality is split into slices / modules. Instead of only grouping by types (e.g. Scripts, Prefabs, Models etc.), the game functionality is grouped by lifetime and features that contains a full vertical slice (Scripts, Models, Materials etc.) for that feature in the *Modules* folder. An example of this is seen below.

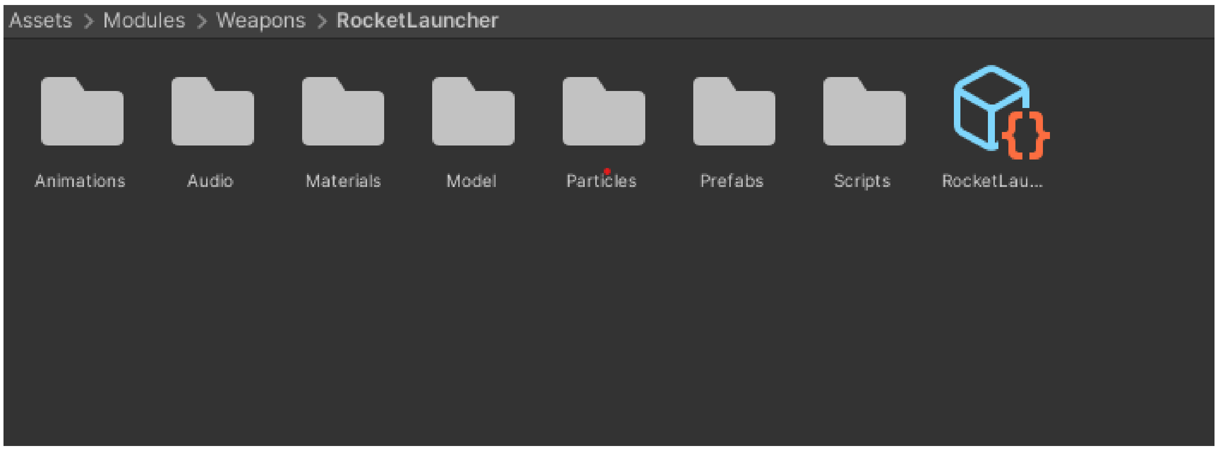


Figure 1: Rocket Launcher Weapon Module

During development it has been strived to follow the following principles:

Modules should work in isolation The functionality of a module should be self-contained as much as possible. This is reflected in how almost all modules exports a single self-contained prefab that exposes the functionality of that module. Some modules technically breaks this principle since they might be composed of other modules, e.g. the ODM (omni directional maneuver) module utilises a prefab from the Weapons module. In this case, knowledge of the inner workings of the external module dependencies should be as little as possible, e.g. by only calling the Prefabs' interfaces or using the event system. An example of this principle being helpful was when originally in this game Grappling Hook swinging functionality was implemented as a weapon "Grappling Gun". After the implemeting it and trying it out, it was not actually fun having to switch weapon to do the grappling. Instead the ODM system was implemented which allows for utilizing two grappling hooks without having to switch weapon. To speed up the development all the grappling hook behaviour could actually be delegated to the original Grappling Gun implementation since it was self-contained, i.e. the ODM prefab just uses the old Grappling Gun prefab through its interface.

Cross-boundary communication should be indirect When indirectly related components needs to communicate, they do it through either events or ScriptableObjects. This is why the MovementController, GasController, WeaponManager etc. broadcasts their state. The state of these different controllers and managers must be read by some UI components such that the player can be made aware of them. However, the subjects that broadcasts these states should not be aware of e.g. UI-related components. The event system is inspired by an article from the course resources and is based on ScriptableObjects. There are two components of the event system - GameEvent and GameEventObserver.

```
// Modules/Events/GameEvent
namespace Modules.Events
{
    [CreateAssetMenu(fileName = "GameEvent", menuName = "Game Event")]
    public class GameEvent : ScriptableObject
    {
        private List<GameEventObserver> _observers = new();

        public void Raise()
        {
            _observers.ForEach(o => o.OnEventRaised(null));
        }

        public void Raise(object data)
        {
            _observers.ForEach(o => o.OnEventRaised(data));
        }

        public void RegisterObserver(GameEventObserver observer)
        {
            _observers.Add(observer);
        }

        public void UnregisterObserver(GameEventObserver observer)
        {
            _observers.Remove(observer);
        }

        public static object NoData()
        {
            return null;
        }
    }
```

Events are ScriptableObjects that act as subjects which other components can subscribe to. The system is quite extensible as creating new events are as easy as just right clicking in the Unity Project view and creating a new GameEvent which some name. Components that have a reference to this GameEvent can then use it to broadcast some data as seen in the example below

```
        private void FixedUpdate()
        {
            gasStateEvent.Raise(new GasStateEvent()
            {
                CurrentLevel = _currentGasLevel,
                ExpenditureRate = gasExpenditureRatePerSecond,
                MaxCapacity = maxGasCapacity
            });
            if (!_active || GasTankEmpty())
            {
                return;
            }

            ExpendGas(gasExpenditureRatePerSecond * Time.fixedDeltaTime);
            PropelOwnerForward();
        }
```

To subscribe to these events, a GameObject must utilize the GameEventObserver component.

```
// Modules/Events/GameEventObserver
    public class GameEventObserver: MonoBehaviour
    {
        [Header("References")]
        [SerializeField]
        private GameEvent gameEvent;
        [SerializeField]
        private OnEvent onEvent;

        private void OnEnable()
        {
            gameEvent.RegisterObserver(this);
        }

        private void OnDisable()
        {
            gameEvent.UnregisterObserver(this);
        }

        public void OnEventRaised(object data)
        {
            onEvent?.Invoke(data);
        }

        public void RegisterCallback(UnityAction<object> cb)
        {
            onEvent.AddListener(cb);
        }

        public void UnregisterCallback(UnityAction<object> cb)
        {
            onEvent.RemoveListener(cb);
        }
    }
```

The responsibility of the GameEventObserver is only registering itself as a observer to the GameEvent and registering callbacks. Example of a callback can be seen below.

```
        private void OnRestoreGas(object data)
        {
            _currentGasLevel = data switch
            {
                float amount when _currentGasLevel + amount <= maxGasCapacity =>
_currentGasLevel + amount,
                float amount when amount > maxGasCapacity => maxGasCapacity,
                 float amount when _currentGasLevel + amount > maxGasCapacity =>
maxGasCapacity,
                _ => _currentGasLevel
            };
        }
```

Weapon System

One of the more interessting parts of the game in terms of implementation is the Weapon System. The weapon system is a module consisting of multiple submodules such as a common module, weapon specific modules and the Weapon Manager module. The requirements for the system was initially:

- It should support more than one weapon
- It should be easily extensible in terms of adding new weapons
- It should support different reload behaviours

Originally the player was meant to have access to a Rocket Launcher, a Grenade Launcher and a Grappling Gun, which is why the system is implemented with the abstractions it has. This was later dropped due to it simply not being fun to use the Grenade Launcher and the grappling gun was repurposed to another system.

Concept

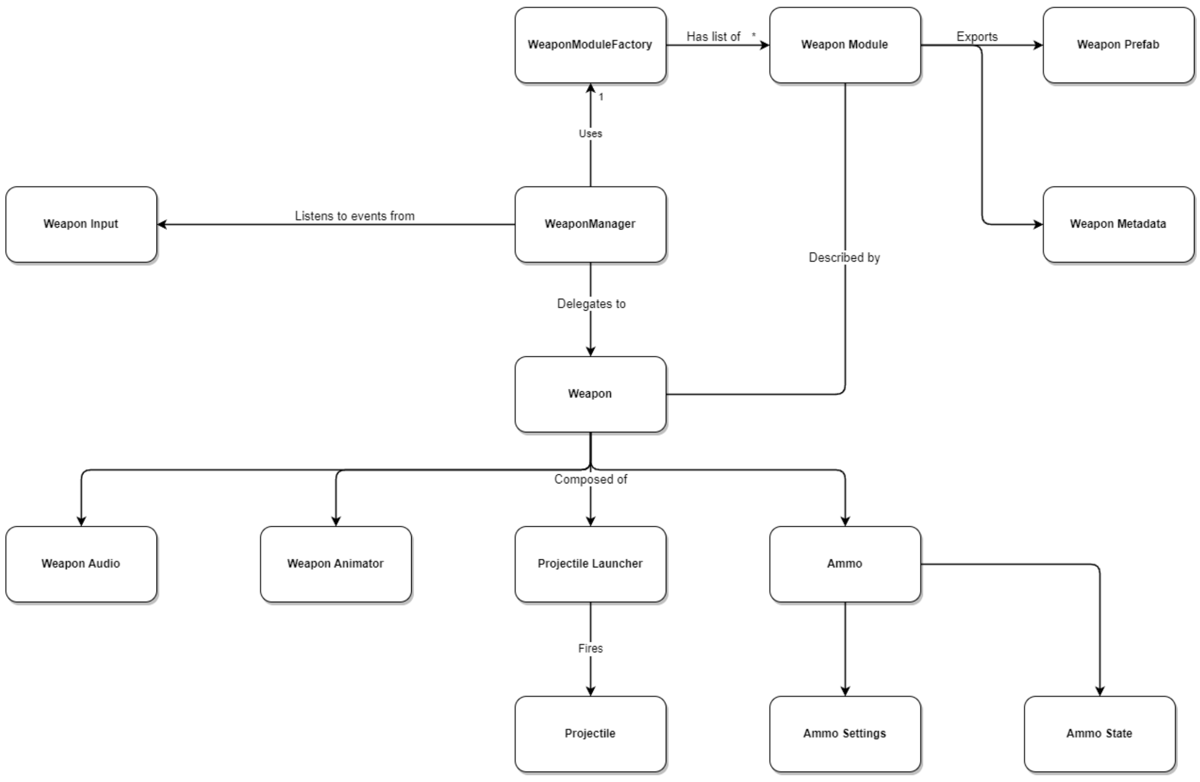


Figure 2: Weapon System general concept

To support more than one weapon a common interface is needed for all the weapons. All weapons in the system consists of the same components, but in different configurations. A common weapon abstraction in form a interface is used, such that the Weapon Manager does not need to know which weapon is actually being using:

```
// Modules/Weapon/WeaponManager/Scripts/WeaponManager
        private void FireCurrentWeapon()
        {
            _currentWeapon.WeaponComponent.FireWeapon();
            EmitFireWeaponEvent();
        }
```

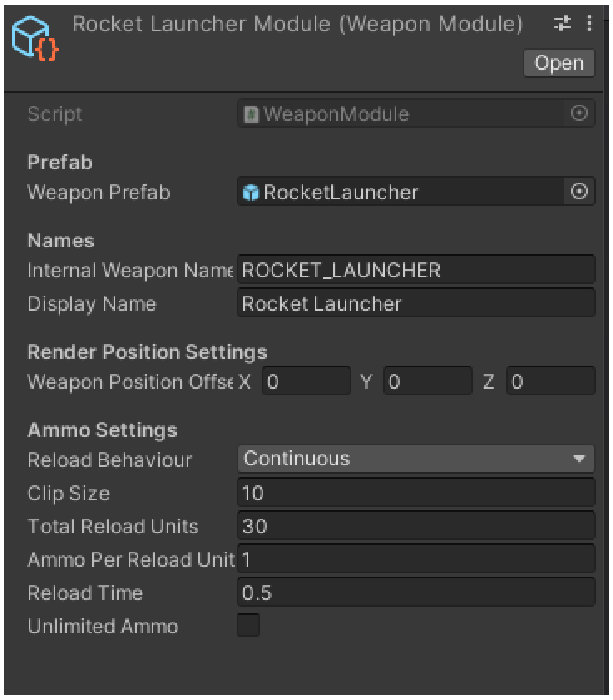
The basic concept is that all weapons are exported via a Weapon Module.

```
namespace Modules.Weapons.Common.Scripts.Weapon
{
    [CreateAssetMenu(menuName = "Weapon Module")]
    public class WeaponModule: ScriptableObject
    {
        [Header("Prefab")]
        public GameObject WeaponPrefab;

        [Header("Names")]
        public string InternalWeaponName;
        public string DisplayName;

        [Header("Render Position Settings")]
        public Vector3 WeaponPositionOffset;


        [Header("Ammo Settings")]
        public ReloadBehaviour ReloadBehaviour;

        public int ClipSize;
        public int TotalReloadUnits;
        public int AmmoPerReloadUnit;
        public float ReloadTime;
        public bool UnlimitedAmmo = false;
    }
}
```

A Weapon Module is a ScriptableObject with has 2 main responsibilities

1. Export the correct prefab
2. Describe metadata about the weapon and behaviour of the weapon

Making Weapon Module a ScriptableObject enables for a nice development experience where creating new weapon for the WeaponManager to use is as simple as making a new weapon prefab with the correct weapon components and creating a WeaponModule asset. Example of a WeaponModule configuration is seen below:



Rocket Launcher Module (Weapon Module)

Script: WeaponModule

Prefab
Weapon Prefab: RocketLauncher

Names
Internal Weapon Name: ROCKET_LAUNCHER
Display Name: Rocket Launcher

Render Position Settings
Weapon Position Offset: X 0, Y 0, Z 0

Ammo Settings
Reload Behaviour: Continuous
Clip Size: 10
Total Reload Units: 30
Ammo Per Reload Unit: 1
Reload Time: 0.5
Unlimited Ammo: ☐

Figure 3: Rocket Launcher Weapon Module

As mentioned earlier all weapons consists of the same components, but in different configurations. This enables creating a baisc base implementation that all weapons can inherit

```
namespace Modules.Weapons.Common.Scripts.Weapon
{
    public class WeaponBase : MonoBehaviour, IWeapon
    {
        public event IWeapon.WeaponEventTrigger ReloadStartedEvent;
        public event IWeapon.WeaponEventTrigger ReloadFinishedEvent;
        protected Ammo.Ammo Ammo;
        private WeaponAudioHandler _audio;
        protected IProjectileLauncher ProjectileLauncher;
        protected WeaponAnimator Animator;

        protected float FireCooldown = 0.7f;
        protected CooldownHandler FireCooldownHandler;

        private Coroutine _reloadRoutine;
        private Coroutine _shootingRoutine;

        protected GameObject Owner;

        [CanBeNull]
        protected Predicate<IWeapon> PreventFirePredicate;
        private void Awake()
        {
            FireCooldownHandler = gameObject.AddComponent<CooldownHandler>();
            _audio = GetComponent<WeaponAudioHandler>();
            ProjectileLauncher = GetComponent<IProjectileLauncher>();
            Ammo = gameObject.AddComponent<Ammo.Ammo>();
            Animator = GetComponent<WeaponAnimator>();

            // NOTE: (mibui 2023-04-20) Ammo is owned by the weapon and they share
    lifetime. Should be fine to not unsubscribe
            Ammo.ReloadStartedEvent += () =>
            {
                if (Ammo.RemainingAmmoCount == 0)
                {
                    return;
                }

                ReloadStartedEvent?.Invoke();
                _audio.PlayReload();
            };

            Ammo.AmmoDepletedEvent += ReloadWeapon;
        }

        public void FireWeapon()
        {
            if (PreventFirePredicate != null && PreventFirePredicate.Invoke(this))
            {
                return;
            }

            if (FireCooldownHandler.CooldownActive())
            {
                return;
            }

            if (Ammo.HasActiveContinuousReload())
            {
                Ammo.InterruptReload();
            }

            Ammo.ConsumeSingleWithActionOrElse(() =>
            {
                FireCooldownHandler.StartCooldown(FireCooldown);
                _shootingRoutine = StartCoroutine(Animator.Fire(FireCooldown));
                ProjectileLauncher.Launch();
                _audio.PlayFireWeapon();
            }, ReloadWeapon);
        }

        public void AlternateFire()
        {
                           ProjectileLauncher.GetActiveProjectiles().ForEach(p  =>
p.TriggerAlternateAction());
        }

        public void ReloadWeapon()
        {
            Ammo.Reload();
        }

        public float GetFireCooldown()
        {
            return FireCooldown;
        }

        public int GetCurrentAmmoCount() => Ammo.CurrentAmmoCount;
        public int GetRemainingAmmoCount() => Ammo.RemainingAmmoCount;

        public void SetAmmoState(AmmoState ammoState)
        {
            Ammo.AmmoState = ammoState;
        }

        public void SetAmmoSettings(AmmoSettings ammoSettings)
        {
            Ammo.AmmoSettings = ammoSettings;
        }

        public void RestoreAmmo(int reloadUnits)
        {
            Ammo.RestoreAmmo(reloadUnits);
        }

        public void SetOwner(GameObject owner)
        {
            Owner = owner;
        }

        public GameObject GetOwner() => Owner;
    }
}
```

The main components of each weapon are:

Ammo

Behaviour relating to reloading and ammo consumption has been encapsulated such that it could be shared between weapons. The Rocket Launcher mechanics in this game has been inspired by Team Fortress 2 where the weapons have different reload behaviours. In this game there are two reload behaviours supported - Continuous and Discrete. A reload is defined as Continuous in this game when ammo can be added to the weapon in small increments such as reloading a rocket launcher rocket by rocket or reloading a shotgun shell by shell. A reload is Discrete in this game when a weapon is either fully reloaded or not reloaded at all after a reload, e.g. reloading an AK47 with a new magazine. The actual settings to be used by Ammo (clip size, reload behaviour, ammo per reload units etc.) is configured in the weapon module. The discrete reload can be handled directly in the reload method as it is an atomic action:

```
// Modules/Weapon/Common/Scripts/Ammo/Ammo
        public void Reload()
        {
            if (_cooldownHandler.CooldownActive())
            {
                ReloadBlockedEvent?.Invoke();
            }

            if (ReloadBehaviour == ReloadBehaviour.Discrete)
            {
                if (GetMissingAmmo() > RemainingAmmoCount)
                {
                    _cooldownHandler.StartCooldown(ReloadTime);
                    AmmoState.CurrentAmmoCount += RemainingAmmoCount;
                    AmmoState.RemainingAmmoCount = 0;
                    return;
                }

                _cooldownHandler.StartCooldown(ReloadTime);
                var missing = GetMissingAmmo();
                AmmoState.CurrentAmmoCount += missing;
                AmmoState.RemainingAmmoCount -= missing;
            }

            if (ReloadBehaviour == ReloadBehaviour.Continuous)
            {
                _reloadingContinuously = true;
            }
        }
```

Cooldown handling is encapsulated in a small utility component that can be used by all other modules that requires cooldowns in any form. Since continuous reloading needs to be handled over multiple frames, it is done in the Update lifecycle hook.

```
        private void Update()
        {
            if (CurrentAmmoCount == 0)
            {
                AmmoDepletedEvent?.Invoke();
            }

            if (!_reloadingContinuously || _cooldownHandler.CooldownActive())
            {
                return;
            }

            if (CurrentAmmoCount == ClipSize)
            {
                InterruptReload();
                return;
            }
            if (AmmoPerReloadUnit > RemainingAmmoCount)
            {
                _cooldownHandler.StartCooldown(ReloadTime);
                AmmoState.CurrentAmmoCount += RemainingAmmoCount;
                AmmoState.RemainingAmmoCount = 0;
                InterruptReload();
                return;
            }

            _cooldownHandler.StartCooldown(ReloadTime);
            AmmoState.CurrentAmmoCount += AmmoPerReloadUnit;
            AmmoState.RemainingAmmoCount -= AmmoPerReloadUnit;
        }
```

Looking back at it, it would probably has been better to implement this as a coroutine as that is exactly what coroutines are for - executing code across multiple frames.

Projectile Launcher

The responsibility of each projectile launcher is as the name implies - launching projectiles. This entails two things:

1. Spawn the projectile
2. Guide / push the projectile towards some destination

One example of a projectile launcher is the PointGuidedProjectileLauncher used by the Rocket Launcher module.

```
namespace Modules.Weapons.Common.Scripts.Launchers
{
    public class PointGuidedProjectileLauncher : MonoBehaviour, IProjectileLauncher
    {
        [Header("References")]
        [SerializeField]
        private GameObject launchPoint;

        [SerializeField]
        private GameObject projectilePrefab;

        [Header("Settings")]
        [SerializeField]
        private LayerMask validDestinationLayers;

        private readonly List<ProjectileInstance> _projectileInstances = new();

        private UnityEngine.Camera _mainCamera;

        private void Awake()
        {
            _mainCamera = UnityEngine.Camera.main;
        }


        public void FixedUpdate()
        {
            ClearDestroyedInstances();
            _projectileInstances.ForEach(i =>
            {
                        MoveProjectileTowardsDestination(i.InstanceRb, i.Projectile,
i.Destination);
            });
        }

        public void Launch()
        {
            Ray ray = RayCastUtil.GetRayToCenterOfScreen(_mainCamera);
                var destination = Physics.Raycast(ray, out RaycastHit hit, 1000f,
validDestinationLayers)
                ? hit.point
                : ray.GetPoint(1000f);

            var projectileObject = Instantiate(projectilePrefab);
            var projectile = projectileObject.GetComponent<IProjectile>();
            var projectileRb = projectileObject.GetComponent<Rigidbody>();
            projectile.Activate(destination);
            projectileObject.transform.position = launchPoint.transform.position;
            projectileObject.transform.forward = _mainCamera.transform.forward;

            _projectileInstances.Add(new ProjectileInstance
            {
                Destination = destination,
                Instance = projectileObject,
                InstanceRb = projectileRb,
                Projectile = projectile
            });
        }

        public List<IProjectile> GetActiveProjectiles()
        {
            return _projectileInstances
                .Select(instance => instance.Projectile)
                .ToList();
        }

        private void ClearDestroyedInstances()
        {
            _projectileInstances.ToList().ForEach(i =>
            {
                if (i.Instance != null)
                {
                    return;
                }

                _projectileInstances.Remove(i);
            });
        }
        private void MoveProjectileTowardsDestination(Rigidbody rb, IProjectile
projectile, Vector3 destination)
        {
            var direction = (destination - rb.transform.position).normalized;
            rb.gameObject.transform.forward = direction;

                rb.AddForce(direction * (projectile.Speed * Time.fixedDeltaTime),
ForceMode.Acceleration);
        }
    }
}
```

The functionality of the PointGuidedProjectileLauncher is simply: spawn projectiles, attain some point the projectiles should travel towards and then keep moving the projectiles towards that point. Other examples of a projectile launcher is the UnguidedProjectileLauncher which is a fire and forget launcher originally used by the Grenade Launcher module.

Projectile

Projectile encapsulates logic about what should happen when the projectile collides another GameObject and how fast should the projectile move after being launched. Examples of projectiles are Rocket, fired by Rocket Launcher, and Hooks, fired by Grappling Guns.

Other components

Each weapon also consists of other components that handles areas such as Animation and Audio. These components are responsible for triggering the correct animation and audioclip when some weapon event has happened such as fire, reload etc.

Weapon Manager

The heart of the weapon system is the Weapon Manager. Its responsibility is to glue the system together by

- Instantiating weapon modules
- Configuring weapons based on the information from the weapon module
- Ensure the weapon is positioned relative to the weaon holder as specified by the weapon module.
- Listen for weapon inputs and delegating the inputs to the weapon instance.

When the Weapon Manager receives a switch weapon input, it will dispose of its current weapon module and instantiate a new one:

```
        private void SwitchWeapon(string weapon)
        {
             if (_currentWeapon.instance != null && _currentWeapon.WeaponComponent !=
null)
            {
                                    _currentWeapon.WeaponComponent.ReloadFinishedEvent  -=
EmitReloadFinishedStateChange;
                                    _currentWeapon.WeaponComponent.ReloadFinishedEvent  -=
EmitReloadStartedStateChange;
                Destroy(_currentWeapon.instance);
            }
            _currentWeaponModule = null;

            var module = moduleFactory.Create(weapon);
            InstantiateModule(module);
            // NOTE: (mibui 2023-04-21) Emit when reload finished to ensure that it is
updated state that gets emitted
            //                          as it takes some time to reload.
                                  _currentWeapon.WeaponComponent!.ReloadFinishedEvent   +=
EmitReloadFinishedStateChange;
                                   _currentWeapon.WeaponComponent!.ReloadStartedEvent   +=
EmitReloadStartedStateChange;
        }
```

Instantiation of a new weapon module consists of

- Instantiate Weapon Module's prefab
- Set owner of the Weapon
- Configure the ammo settings for the weapon

```
        private void InstantiateModule(WeaponModule module)
        {
            _currentWeaponModule = module;

            var weaponInstance = Instantiate(
                _currentWeaponModule switch
                {
                    null => throw new ArgumentException("No Weapon Module"),
                    WeaponModule wm => wm.WeaponPrefab
                },
                weaponHolder.transform, false);

            var weaponComponent = weaponInstance.GetComponent<IWeapon>();
            weaponComponent.SetOwner(owner);

            _currentWeapon.instance = weaponInstance;
            _currentWeapon.WeaponComponent = weaponComponent;

            ConfigureAmmoForModule(module);
        }
```

Weapon input is received through events and each weapon needs a owner. It has been designed this way such that in theory the weapon system could also be used by AI's and other characters.

The Weapon Manager gets the different Weapon Modules through the WeaponModuleFactory, which is a ScriptableObject with a list of Weapon Modules.

```
namespace Modules.Weapons.Common.Scripts.Weapon
{
    [CreateAssetMenu]
    public class WeaponModuleFactory : ScriptableObject
    {
        public List<WeaponModule> Modules;

        public WeaponModule Create(string weapon)
        {
            var module = Modules.First(m => m.InternalWeaponName == weapon)
                .ThrowIfNull(() => new ArgumentException("Module is not supported"));

            return module;
        }

        public WeaponModule GetDefault()
        {
                                  return  Modules.First(m  =>  m.InternalWeaponName  ==
SupportedWeaponModules.RocketLauncher);
        }
    }
}
```

This allows the Weapon Manager to instantiate new weapon modules without having the modify the Weapon Manager each time a new weapon gets added to the game. To register a new weapon module you simply have to add it in the inspector:

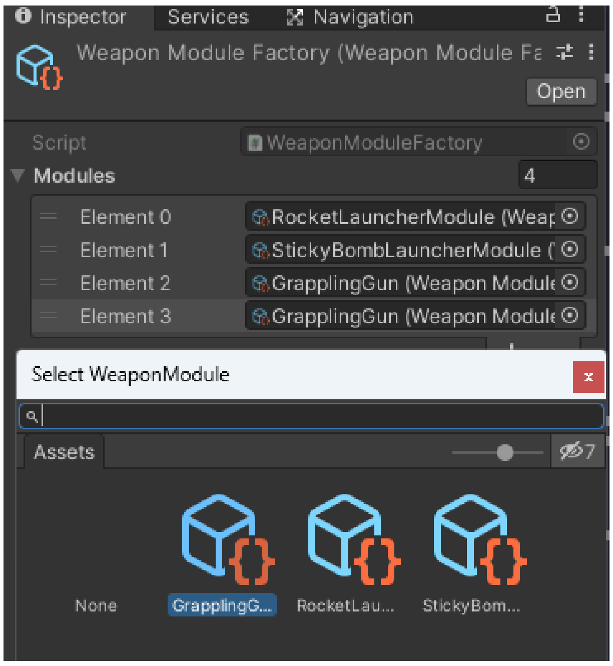


Figure 4: Weapon Module Factory inspector