Implementation Report

Enemy State Machine

As the enemies in the game have very fixed logic the state pattern lends itself well to implementing a controller for them. The state machine exists as an `EnemyState` instance inside the `EnemyController` class which is then updated by the class:

```
using System;
using System.Collections.Generic;


using UnityEngine;
namespace Enemy {
    public abstract class EnemyState : IEnemyState {
        protected readonly EnemyController _controller;
        protected readonly SpriteRenderer _spriteRenderer;
        protected readonly EnemyManager _enemyManager;
        protected readonly EnemyStateFactory _enemyStateFactory;

        public EnemyState(EnemyController controller, SpriteRenderer spriteRenderer,
EnemyManager enemyManager, EnemyStateFactory enemyStateFactory) {
            _controller = controller;
            _spriteRenderer = spriteRenderer;
            _enemyManager = enemyManager;
            _enemyStateFactory = enemyStateFactory;
        }

        public abstract void Start();
        public abstract void FixedUpdate();
        public abstract void Update();
        public abstract void CheckTransitions();
        public abstract void Exit();

        public void SwitchState(EnemyState state) {
            _controller.State?.Exit();
            _controller.State = state;
            _controller.State.Start();
        }

        public class EnemyStateFactory {

            private Dictionary<Type, EnemyState> _states;

            public EnemyStateFactory(EnemyController controller, SpriteRenderer renderer,
EnemyManager manager) {
                _states = new Dictionary<Type, EnemyState>() {
                { typeof(EnemyIdleState), new EnemyIdleState(controller, renderer,
manager, this) },
                { typeof(EnemyPatrolState), new EnemyPatrolState(controller, renderer,
manager, this) },
                { typeof(EnemyChaseState), new EnemyChaseState(controller, renderer,
manager, this) },
                { typeof(EnemyAttackState), new EnemyAttackState(controller, renderer,
manager, this) },
            };
            }

            public void ValidateStates() {
                foreach (EnemyState state in _states.Values) {
                    Debug.Log($"{state.GetType()} is valid: {(state._enemyStateFactory ==
null ? "no" : "yes")}");
                }
            }

            public T State<T>() where T : EnemyState {
                if (!_states.ContainsKey(typeof(T))) {
                    Debug.Log($"Could not find state: {typeof(T)}");
                    return _states[typeof(EnemyIdleState)] as T;
                }
                return _states[typeof(T)] as T;
            }
        }
    }
}
```

In the `EnemyController` class is responsible for calling the concrete implementation of the abstract methods defined in `EnemyState` along with managing some other references, providing some properties for the states to use and rotating the enemy as the `NavMeshAgent` does not work with 2D rotation:

```
using System.Collections.Generic;
using System.Linq;

using UnityEngine;
using UnityEngine.AI;

using Utilities;

namespace Enemy {
    public class EnemyController : MonoBehaviour {

        [Header("Component References")]
        [SerializeField] private NavMeshAgent _agent;
        [SerializeField] private EnemyManager _enemyManager;

        [Header("Enemy Parameters")]
        [SerializeField] private float _maxSpeed = 200f;
        [SerializeField] private float _maxAcceleration = 60f;
        [SerializeField] private float _maxTurnSpeed = 5f;
        [SerializeField] private float _chaseRange = 10f;
        [SerializeField] private float _attackRange = 5f;
        [SerializeField] private float _fireRate = 1f;
        [SerializeField] private float _damage = 1f;

        [Header("Gameplay Variables")]
        [SerializeField] private CountDownTimer _attackTimer;
        private EnemyState _state;
        private EnemyState.EnemyStateFactory _stateFactory;

        public EnemyState State { get { return _state; } set { _state = value; } }
        public NavMeshAgent Agent { get { return _agent; } }
        public CountDownTimer AttackTimer { get { return _attackTimer; } }
        public float Damage => _damage;
        public bool InChaseRange => HasTarget &&
Vector3.Distance(_enemyManager.Target.OrNull()?.position ?? new Vector3(Mathf.Infinity,
Mathf.Infinity), transform.position) <= _chaseRange;
        public bool InAttackRange => HasTarget &&
Vector3.Distance(_enemyManager.Target.OrNull()?.position ?? new Vector3(Mathf.Infinity,
Mathf.Infinity), transform.position) <= _attackRange;
        public bool HasTarget => _enemyManager.Target != null;

        public void Awake() {
            _agent = GetComponent<NavMeshAgent>();
            _agent.acceleration = _maxAcceleration;
            _agent.speed = _maxSpeed;
            _agent.updateUpAxis = false;
            _agent.updateRotation = false;
            _agent.stoppingDistance = _attackRange - 0.5f;
            _enemyManager = transform.parent.GetComponent<EnemyManager>();
            _attackTimer = new CountDownTimer(_fireRate);
        }

        public void Start() {
            _stateFactory = new EnemyState.EnemyStateFactory(this,
GetComponent<SpriteRenderer>(), _enemyManager);
            _state = _stateFactory.State<EnemyPatrolState>();
            if (_state == null) {
                Debug.LogError("State was not initialised!");
            }
            _state?.Start();
        }

        public void Update() {
            _state?.Update();
            _state?.CheckTransitions();
            UpdateRotation();
        }

        public void FixedUpdate() {
            _state?.FixedUpdate();
            _attackTimer.Update(Time.fixedDeltaTime);
        }
        public void UpdateRotation() {
            if (!HasTarget) {
                return;
            }
            float angle = Vector2.SignedAngle((Vector2)(_enemyManager.Target.position -
transform.position), Vector2.up);
            Quaternion rotation = Quaternion.AngleAxis(angle, Vector3.back);
            transform.rotation = Quaternion.Slerp(transform.rotation, rotation,
Time.fixedDeltaTime * _maxTurnSpeed);
        }
    }
}
```

Boss Attack Strategies

The boss attacks in the game are stored as a list of `BossAttacks` which is an abstract class inheriting `ScriptableObject` and providing an abstract `Attack(Transform origin)` method that concrete implementations must implement.

```
using UnityEngine;
namespace Boss {

    public abstract class BossAttack : ScriptableObject {
        public float Cooldown = 1f;
        public abstract void Attack(Transform origin);
    }
}
```

Then using this as a base I can create the concrete implementations of the attacks and choose which components I should add and also add any prefab references statically in the `ScriptableObject`:

```
using System.Linq;

using UnityEngine;

namespace Boss {
    public class ArcProjectileController : MonoBehaviour {
        [SerializeField] private float _speed;
        [SerializeField] private float _radius;
        [SerializeField] private float _damage;
        [SerializeField] private Vector3 _target;
        [SerializeField] private Vector3 _linearPosition;
        [SerializeField] private Vector3 _initialPosition;
        [SerializeField] private float _progress = 0f;
        [SerializeField] private float _height;
        [SerializeField] private Rigidbody2D _rb2D;

        private void Awake() {
            _rb2D = GetComponent<Rigidbody2D>();
            _radius = GetComponent<SpriteRenderer>().bounds.extents.x;
        }
        public void Init(float speed, float damage, float height, Vector3 target) {
            _speed = speed;
            _damage = damage;
            _target = target;
            _height = height;
            _linearPosition = transform.position;
            _initialPosition = transform.position;
        }

        public void FixedUpdate() {
            if (Vector2.Distance(_target, transform.position) < _radius) {
                Physics2D.OverlapCircleAll(transform.position, _radius,
Globals.Instance.PlayerLayer)
                    .Where((Collider2D entity) =>
entity.gameObject.HasComponent<PlayerController>())
                    .FirstOrDefault().OrNull()?
                    .GetComponent<Health>().OrNull()?
                    .Damage(_damage);
                GameObject hitParticles = Instantiate(Assets.Instance.HitParticles,
transform.position, Quaternion.LookRotation(-transform.up));
                Destroy(hitParticles, 1f);
                Destroy(gameObject);
            } else {
                //TODO: Fix this to use an arc
                _progress = Mathf.Clamp01(Vector2.Distance(_linearPosition, _target) /
Vector2.Distance(_initialPosition, _target));
                _linearPosition = Vector3.MoveTowards(_linearPosition, _target,
Time.fixedDeltaTime * _speed);
                _rb2D.MovePosition(_linearPosition + _height * Mathf.Sin(_progress *
Mathf.PI) * Vector3.up);
            }
        }
    }

}
```

Saving Player Settings

Saving player settings is handled by serialising and deserialising the `Settings` class to and from a JSON format, the `SettingsManager` class which handles callbacks for the sliders to change the setting values and also saving and loading data:

```
using System;
using System.Collections;
using System.IO;

using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using Utilities;

public class SettingsManager : MonoBehaviour {
    [SerializeField] private Settings _currentSettings;
    public Settings CurrentSettings => _currentSettings;
    [SerializeField] private Toggle _tutorialToggle;
    [SerializeField] private Slider _globalVolumeSlider;
    [SerializeField] private Slider _sfxVolumeSlider;
    [SerializeField] private Slider _bgmVolumeSlider;
    [SerializeField] private bool _inMainMenu = false;
    [SerializeField] private AudioMixer _audioMixer;

    private Coroutine _saveCoroutine = null;

    private string _filePath;

    private void Awake() {
        FindFirstObjectByType<WeaponAugmentUIManager>();
        _filePath = Path.Combine(Application.dataPath, "Settings.data");
        if (File.Exists(_filePath)) {
            _currentSettings = JsonUtility.FromJson<Settings>
(File.ReadAllText(_filePath));
            _currentSettings ??= Settings.Defaults; // Can't compress this as unity is
initialising Settings
        } else {
            _currentSettings = Settings.Defaults;
        }
        _inMainMenu = SceneManager.GetActiveScene().buildIndex == 0;
        if (!_inMainMenu) {
            GetComponent<UIMover>().OnCloseFinish += Save;
        }
    }

    private void Start() {
        _tutorialToggle.onValueChanged.AddListener((bool value) => {
            _currentSettings.TutorialMode = value;
            ApplySettings();
        });
        _globalVolumeSlider.onValueChanged.AddListener((float value) => {
            _currentSettings.GlobalVolume = VolumeRemap(value);
            ApplySettings();
        });
        _sfxVolumeSlider.onValueChanged.AddListener((float value) => {
            _currentSettings.SFXVolume = VolumeRemap(value);
            ApplySettings();
        });
        _bgmVolumeSlider.onValueChanged.AddListener((float value) => {
            _currentSettings.BGMVolume = VolumeRemap(value);
            ApplySettings();
        });
        LoadSettings();
    }
    private void ApplySettings() {
        if (_inMainMenu) {
            Settings.ApplySettings(_currentSettings, true, _audioMixer);
        } else {
            Settings.ApplySettings(_currentSettings);
        }
    }

    private float VolumeRemap(float value) {
        return 20f * Mathf.Log10(value);
    }

    private float ReverseVolumeRemap(float value) {
        return Mathf.Pow(10, value / 20f);
    }

    public void Save() {
        if (_saveCoroutine != null) {
            StopCoroutine(_saveCoroutine);
            _saveCoroutine = null;
        }
        _saveCoroutine = StartCoroutine(SaveData());
    }

    private IEnumerator SaveData() {
        string json = JsonUtility.ToJson(_currentSettings, true);
        yield return Yielders.WaitForFixedUpdate;
        File.WriteAllText(_filePath, json);
        _saveCoroutine = null;
    }

    public void LoadSettings() {
        _globalVolumeSlider.value = ReverseVolumeRemap(_currentSettings.GlobalVolume);
        _sfxVolumeSlider.value = ReverseVolumeRemap(_currentSettings.SFXVolume);
        _bgmVolumeSlider.value = ReverseVolumeRemap(_currentSettings.BGMVolume);
        _tutorialToggle.isOn = _currentSettings.TutorialMode;
    }
}
```

The `Settings` class is a simple class to hold volume and tutorial settings with a few fields and a static `ApplySettings()` method that is overloaded so it can be used in both the main menu and in game as the `Globals` singleton providing a reference to the `AudioMixer` is not present in the main menu.

Player Attacks

Player attacks use the same strategy pattern solution as boss attacks however they are given a class allowing the player to have multiple different attacks based on available augment slots but is able to use all augments of the same type at once:

In this case concrete implementations of the `ProjectileSpawnStrategy ScriptableObject` have to define a `Fire` method where they create instances of a bullet prefab and attach relevant components:

```
using ProjectileComponents;

using UnityEngine;

[CreateAssetMenu(fileName = "Shotgun Spawn Strategy", menuName = "Projectile Spawn
Strategy/Shotgun")]
public class ShotgunSpawnStrategy : ProjectileSpawnStrategy {
    public GameObject Projectile;

    public ShotgunSpawnStrategy(GameObject projectile) {
        Projectile = projectile;
    }

    public float Count;
    public float SpreadAngle;

    public override void Fire(Transform origin) {
        float startAngle = Vector2.SignedAngle(origin.up, Vector3.up);
        float halfSpread = SpreadAngle * 0.5f;
        for (float angle = startAngle - halfSpread; angle <= startAngle + halfSpread;
angle += SpreadAngle / Count) {
            GameObject projectileInstance = Instantiate(Projectile, origin.position +
(Vector3) (Helpers.FromRadians(angle * Mathf.Deg2Rad) * 2f), Quaternion.AngleAxis(angle,
Vector3.back));
            projectileInstance.GetOrAddComponent<AutoDestroy>().Duration = Duration;
            projectileInstance.GetOrAddComponent<ProjectileMover>().Speed = Speed;
            projectileInstance.GetOrAddComponent<PlayerProjectile>();
            projectileInstance.GetOrAddComponent<EntityDamager>().Init(DamageFilter.Enemy,
Damage);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using UnityEngine;

public class ProjectileSpawnerManager : MonoBehaviour {
    [SerializeField] private Dictionary<ProjectileSpawnStrategyType, Transform[]>
_sockets;
    [SerializeField] private Dictionary<ProjectileSpawnStrategyType,
List<ProjectileSpawner>> _projectileSpawners;

    private void Awake() {
        _sockets = new Dictionary<ProjectileSpawnStrategyType, Transform[]>();
        _projectileSpawners = new Dictionary<ProjectileSpawnStrategyType,
List<ProjectileSpawner>>();
        foreach (ProjectileSpawnStrategyType type in
Enum.GetValues(typeof(ProjectileSpawnStrategyType))) {
            _projectileSpawners.Add(type, new List<ProjectileSpawner>());
        }
        InitSockets<LightSocket>(ProjectileSpawnStrategyType.LIGHT);
        InitSockets<HeavySocket>(ProjectileSpawnStrategyType.HEAVY);
        InitSockets<EliteSocket>(ProjectileSpawnStrategyType.ELITE);
    }

    private void InitSockets<T>(ProjectileSpawnStrategyType strategyType) where T :
Component {
        Transform socketRoot = FindAnyObjectByType<T>().OrNull()?.transform ?? null;
        if (socketRoot == null) {
            Debug.LogWarning("Could not find socket of type: " + typeof(T));
            return;
        }
        List<Transform> sockets = new List<Transform>();
        for (int i = 0; i < socketRoot.childCount; i++) {
            sockets.Add(socketRoot.GetChild(i));
        }
        _sockets.Add(strategyType, sockets.ToArray());
    }

    public void TryAddSpawner(ProjectileSpawnStrategy spawnStrategy) {
        _sockets.TryGetValue(spawnStrategy.Type, out Transform[] sockets);
        if (sockets == null) {
            Debug.LogWarning("Could not find sockets for type: " + spawnStrategy.Type);
            return;
        }
        foreach (Transform socket in sockets) {
            if (socket.childCount == 0) {
                GameObject firePoint = new GameObject($"{spawnStrategy.Type}");
                firePoint.transform.parent = socket;
                firePoint.transform.SetLocalPositionAndRotation(Vector3.zero,
Quaternion.identity);
                _projectileSpawners[spawnStrategy.Type].Add(new
ProjectileSpawner(spawnStrategy, firePoint.transform));
                Globals.Instance.AddMoney(-spawnStrategy.Cost);
                return;
            }
        }
    }

    public bool Fire(ProjectileSpawnStrategyType type) {
        foreach (ProjectileSpawner projectileSpawner in _projectileSpawners[type]) {
            projectileSpawner.Fire();
        }
        return _projectileSpawners[type].Count > 0;
    }
}
```

The class stores references to all `ProjectileSpawner` instances in a dictionary keyed by their `ProjectileSpawnStrategyType` which internally contains a reference to the spawn point `Transform` and `ProjectileSpawnStrategy` to use when its `Fire()` method is called:

```
using System;
using System.Collections.Generic;
using UnityEngine;


[Serializable]
public class ProjectileSpawner {

    [SerializeField] private ProjectileSpawnStrategy _spawnStrategy;
    [SerializeField] private Transform _firePoint;

    public ProjectileSpawnStrategy Strategy { get => _spawnStrategy; }

    public ProjectileSpawner(ProjectileSpawnStrategy spawnStrategy, Transform firePoint) {
        _spawnStrategy = spawnStrategy;
        _firePoint = firePoint;
    }

    public void Fire() {
        _spawnStrategy.Fire(_firePoint);
    }
}
```

Sound

Sound effects are implemented through the use of a `SFXEmitter` class that is attached as a component to `GameObjects`, the sound effects used by the emitter are then specified using an enum array in the editor where the `SoundManager` instance in the `Globals` singleton then provides all the necessary `AudioClip` instances in the `Start()` lifetime of the `SFXEmitter`:

```
using System;
using System.Collections;
using System.Collections.Generic;

using UnityEngine;

using Utilities;

public class SFXEmitter : MonoBehaviour {
    [SerializeField] private Dictionary<SoundEffectType, AudioSource> _sources;
    [SerializeField] private SoundEffectType[] _effects;
    const float PITCH_BEND_AMOUNT = 10f;

    private void Start() {
        _sources = new Dictionary<SoundEffectType, AudioSource>();
        foreach (SoundEffectType type in _effects) {
            AudioSource audioSource = gameObject.AddComponent<AudioSource>();
            _sources.Add(type, audioSource);
            audioSource.outputAudioMixerGroup = Globals.Instance.SoundManager.SFX;
            audioSource.clip = Globals.Instance.SoundManager.GetClip(type);
        }
    }

    public void Play(SoundEffectType soundEffect) {
        if (_sources.ContainsKey(soundEffect)) {
            _sources[soundEffect].Play();
        }
    }

    public void Play(SoundEffectType soundEffect, float pitchRandomisation) {
        if (_sources[soundEffect].clip == null) {
            _sources[soundEffect].pitch += pitchRandomisation / PITCH_BEND_AMOUNT;
            _sources[soundEffect].Play();
            StartCoroutine(ResetClip(soundEffect));
        }
    }

    private IEnumerator ResetClip(SoundEffectType soundEffect) {
        while (_sources[soundEffect].isPlaying) {
            yield return Yielders.WaitForSeconds(0.1f);
        }
        _sources[soundEffect].pitch = 1;
    }
}
```

Extensions

I created a set of extension methods to ease development, one of the key ones being an extension to `Unity.Object` allowing me to use null propagation, this cannot normally be done as when an object is null in Unity it is just marked for deletion rather than truly null and the boolean comparison is operator overloaded so you use `if (component) { // Logic }` syntax

```
/// <summary>
/// Gets, or adds if doesn't contain a component
/// </summary>
/// <typeparam name="T">Component Type</typeparam>
/// <param name="gameObject">GameObject to get component from</param>
/// <returns>Component</returns>
public static T GetOrAddComponent<T>(this GameObject gameObject) where T : Component {
    T component = gameObject.GetComponent<T>();
    if (!component) {
        component = gameObject.AddComponent<T>();
    }
    return component;
}

/// <summary>
/// Returns true if a GameObject has a component of type T
/// </summary>
/// <typeparam name="T">Component Type</typeparam>
/// <param name="gameObject">GameObject to check for component on</param>
/// <returns>If the component is present</returns>
```

```
public static bool HasComponent<T>(this GameObject gameObject) where T : Component {
    return gameObject.GetComponent<T>() != null;
}

/// <summary>
/// Allows for use of null propogation on Unity Components as Unity uses
/// null as 'marked for destroying', for example:
/// <code>
/// float value = GetComponent&lt;MagicType&gt;().OrNull&lt;MagicType&gt;
()?.MagicFloatField ?? _defaultMagicFloatValue;
/// </code>
/// </summary>
/// <typeparam name="T">Type of UnityObject to check for being actually bull</typeparam>
/// <param name="obj">Object to check for null reference on</param>
/// <returns>T or null if marked as null</returns>
public static T OrNull<T>(this T obj) where T : UnityEngine.Object => obj ? obj : null;

///<summary>
///Normalises a Vector3 if its magnitude is larger than one
///</summary>
///<param name="vector">Vector to clamp</param>
public static void ClampToNormalised(this Vector3 vector) {
    if (vector.magnitude > 1f) {
        vector.Normalize();
    }
}

///<summary>
///Modifies the specified component(s) of a vector
///</summary>
///<param name="vector">Vector to modifiy</param>
///<param name="x">New x value if specified</param>
///<param name="y">New y value if specified</param>
///<param name="z">New z value if specified</param>
///<returns>Modified vector</returns>
public static Vector3 With(this Vector3 vector, float? x, float? y, float? z) {
    return new Vector3(x ?? vector.x, y ?? vector.y, z ?? vector.z);

}

///<summary>
///Adds to the specified component(s) of a vector
///</summary>
///<param name="vector">Vector to modifiy</param>
///<param name="x">Increase in x value if specified</param>
///<param name="y">Increase in y value if specified</param>
///<param name="z">Increase in z value if specified</param>
///<returns>Modified vector</returns>
public static Vector3 Add(this Vector3 vector, float? x = 0, float? y = 0, float? z = 0) {
    return new Vector3(vector.x + (x ?? 0f), vector.y + (y ?? 0f), vector.z + (z ?? 0f));
}

///<summary>
```

```
///Adds to the specified component(s) of a vector
///</summary>
///<param name="vector">Vector to modifiy</param>
///<param name="x">Increase in x value if specified</param>
///<param name="y">Increase in y value if specified</param>
///<param name="z">Increase in z value if specified</param>
///<returns>Modified vector</returns>
public static Vector2 Multiply(this Vector2 vector, float? x = 1f, float? y = 1f) {
    return new Vector2(vector.x * (x ?? 1f), vector.y * (y ?? 1f));
}

public static Vector2 ClampMagnitude(this Vector2 vector, float magnitude) {
    if (vector.sqrMagnitude > (magnitude * magnitude)) {
        vector.Normalize();
        vector *= magnitude;
    }
    return vector;
}

///<summary>
///Changes the colour of the material on the provided SpriteRenderer for the specified
time
///using a coroutine that must have the MonoBehaviour to attach the coroutine to
///</summary>
///<param name="spriteRenderer">SpriteRenderer to change material colour of</param>
///<param name="colour">Colour to change SpriteRenderer material to</param>
///<param name="time">Time until colour changes back</param>
///<param name="monoBehaviour">MonoBehaviour to start coroutine on</param>
public static void FlashColour(this SpriteRenderer spriteRenderer, Color colour, float
time, MonoBehaviour monoBehaviour) {
    if (monoBehaviour.OrNull() == null) {
        Debug.Log("Provided MonoBehaviour was null!");
        return;
    }
    monoBehaviour.StartCoroutine(Flash(spriteRenderer, colour, time));
}

public static void FlashColour(this SpriteRenderer spriteRenderer, Color flashColour,
Color originalColour, float time, MonoBehaviour monoBehaviour) {
    if (monoBehaviour.OrNull() == null) {
        Debug.Log("Provided MonoBehaviour was null!");
        return;
    }
    monoBehaviour.StartCoroutine(Flash(spriteRenderer, flashColour, originalColour,
time));
}


private static IEnumerator Flash(SpriteRenderer spriteRenderer, Color colour, float time)
{
    Color original = spriteRenderer.color;
    spriteRenderer.color = colour;
    yield return Yielders.WaitForSeconds(time);
    spriteRenderer.color = original;
}

private static IEnumerator Flash(SpriteRenderer spriteRenderer, Color colour, Color
originalColour, float time) {
    spriteRenderer.color = colour;
    yield return Yielders.WaitForSeconds(time);
    spriteRenderer.color = originalColour;
}
```