<u>Introduction:</u>

In this Demo project, the main idea is to play with different aspects of AI techniques in a video game, apart from standard functionalities. On one hand we will explain the procedural part of terrain generation using Perlin Noise, how to parallelize this process to obtain speed benefits, and on the other hand the usage of Finite State Machines. These FSM will be used in two forms, one using the functionalities that Unity can bring us, and the other one is a custom FSM coded by hand to play with.
Regarding the terrain generation, this process will be divided into several parts. Firstly, the creation of a fixed terrain chunk size. Then, the also procedural instantiation of terrain objects like trees and bushes, and the ability to increase or decrease the detail of the mesh depending on the distance. Apart from this, as the terrain can be infinite, an optimization part of disabling and enabling different parts of the terrain is implemented if the player is too far away, so we don't need to have the entire map loaded at one.
As video games code structure is not exactly the same as backend projects for example, the UML diagrams that will be used in this document are approximations to have a better overview of what is going on in the code. Also, the mathematical details and the behavior of FSM will not be introduced in a deep detail.

In terms of gameplay, it is simple. There is no real goal, the player will have a companion that will try to defend it, and random enemies will spawn nearby. The behavior of the companion and the enemies will be explained later. Even though the player can lose life, it will not die. The idea is to reach a Cabin safety, but also the game will not finish as this is just a prototype to play with.

<u>Perlin Noise:</u>

Perlin Noise is a random but organic noise widely used in video games. To create the Perlin Noise values, we will use the functionality that Unity can give in the Mathematical libraries that it has.
In this project, perlin noise will be used to generate the terrain of the game. The main idea is to apply the values of the 2D array to the Y values of the triangle mesh. In this way, we can control the outline of the terrain. To have a better clear idea of the difference between regular noise vs perlin noise, we can have a look at Figure 1.

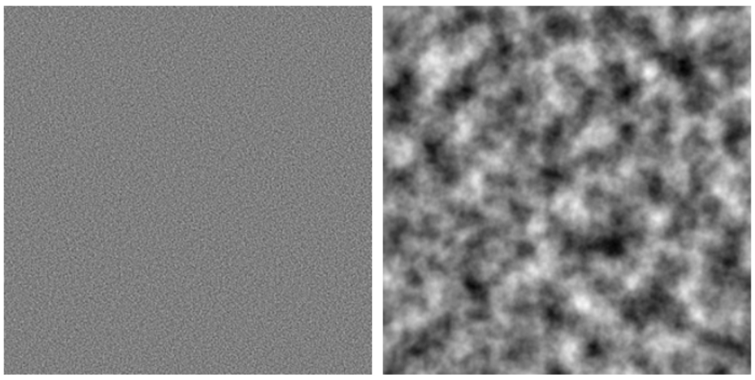


Figure 1. Difference of regular noise vs perlin noise.

In 2 Dimensions, Perlin Noise can be seen as wave functions, also called Octaves. These octaves represent the details. If we have just one octave, the result will be too smooth. We want to increase the detail by adding more octaves, so the wave function will have a less and more natural shape.

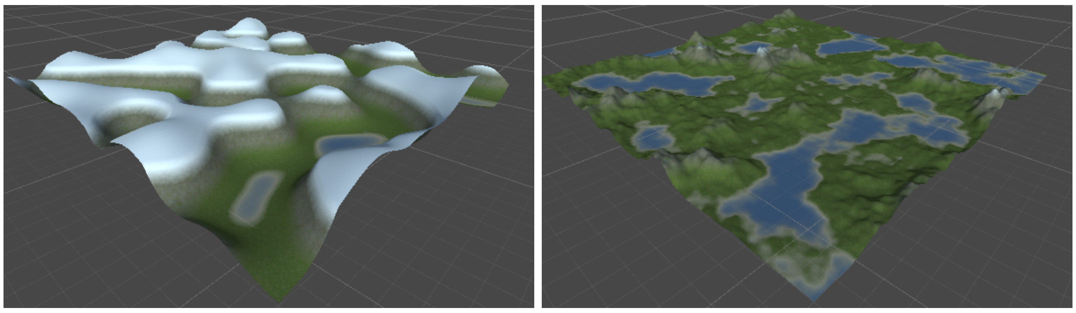


Figure 2 and 3. One octave vs five octaves result.

In the previous figure, we can see a visual explanation of the effect of adding several octaves. We have other parameters for this generation. Lacunarity controls the importance of small features. The more lacunarity we have, the more irregular the terrain will be, and vice versa.

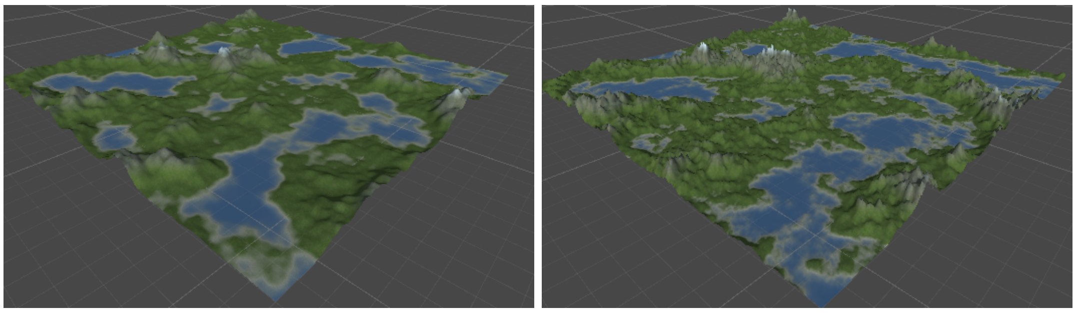


Figure 3 and 4. Difference between lacunarity 2 and 3.

Using the same figure as before, Figure 3, we will increase the lacunarity value from 2 to 3, and we can see that the map is more irregular, as it was expected. Another important parameter is Persistance, which will determine how much each octave contributes to the overall structure of the noise map.

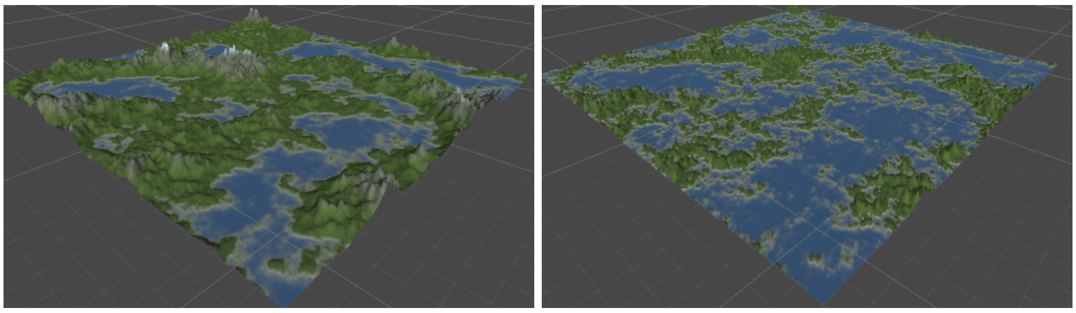


Figure 4 and 5. Difference between persistance 0.5 and 0.75.

Again, doing some tests in our map, we can see that increasing the persistance the octaves will have a greater contribution, thus making the details have a bigger impact in the result.

Then we have other parameters that will be explained in the video. All of this is in **HeightNoise.asset,** under PCGAssets/Data folder. In this folder we will also have the asset **DefaultTexture.asset**. This asset will be used to apply colour to the mesh, depending on the height of the values. This will start with a blue colour to height zero, scaling up to white (representing snow).

Another easy thing to implement is a border limitation. A second array of noise values between [-1, 1] can be subtracted to the original noise map to obtain a border limitation. In order to modify the strongness of the falloff, we can also use a sigmoidal function to make the numbers in the border with higher density than those in the center. In this case the function used will be:

$$f(x) = \frac{x^a}{x^a + (b - bx)^a}$$

Equation 1

The parameter a in this function can control the amount of values near to zero, and b controls the slide. With values as a = 3 and b = 2.2 we can have a shape like this:

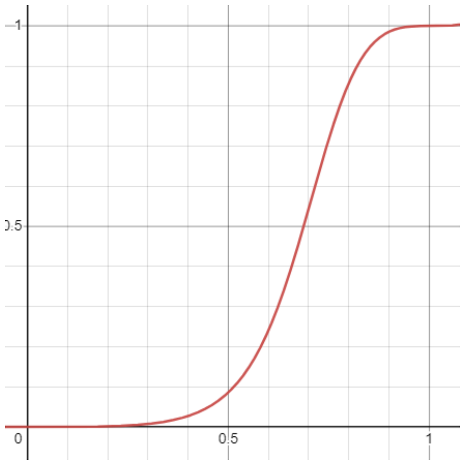


Figure 6. Curve of Function 1 with a=3, b=2.2 values.

Once this falloff map is implemented and subtracted from the noise map, we can obtain this result:

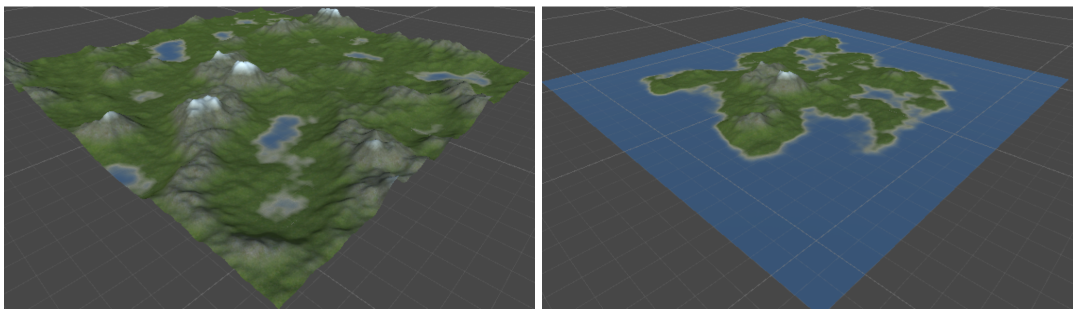


Figure 7. Comparison between not using falloff and applying it.

Level of Detail:

As we can easily manage to create the information of the mesh, we can easily implement a way to change the detail of the mesh depending on the position of the player. The main idea behind this is to add an offset when we iterate over the vertices of our mesh. Basically the maximum number of vertices selected for this project is 241. Thus, our chunk size will be 241 x 241. With this, we can change the detail of the mesh iterating with a step of 1, 2, 4, 6, 8, 10 and 12 the array of vertices of the map, since the step has to be a factor of size - 1 (240) because we are starting at position zero.

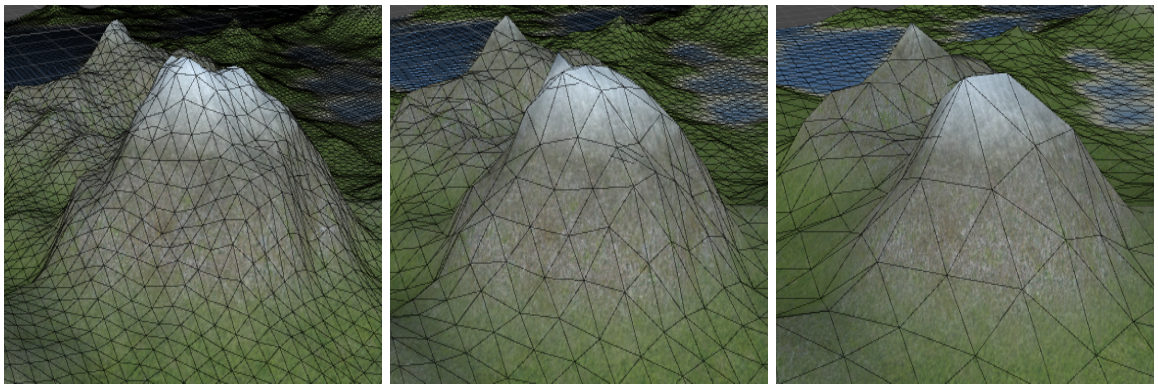
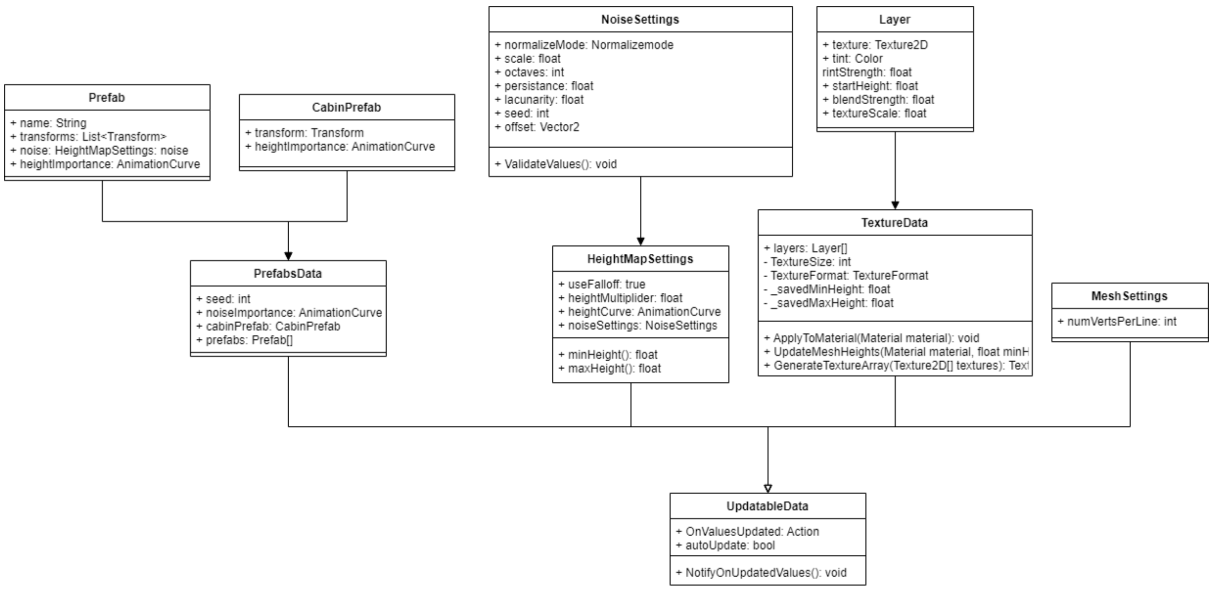


Figure 8. Differences between step 1, 2 and 4.

Code Structure:

Here we have a UML diagram explaining a little bit the structure of the data that can be modified in the inspector to obtain different results. This will be explained in the video, but regarding the data that can be modified in the editor, we have this structure:



UML 1. UpdatableData structure.

<u>Scene assets:</u>

Now that the most important parts of the terrain are explained. We can jump to the instantiation of the prefabs of the game. For the prefabs, we will have different height importances and noises so we can play with the population of trees and bushes for example, in the same way we can play with the terrain shape.
For the positions, the points of spawning will be the first point of every triangle in the mesh. With this, the more detailed the mesh, the more populated will be, and vice versa.

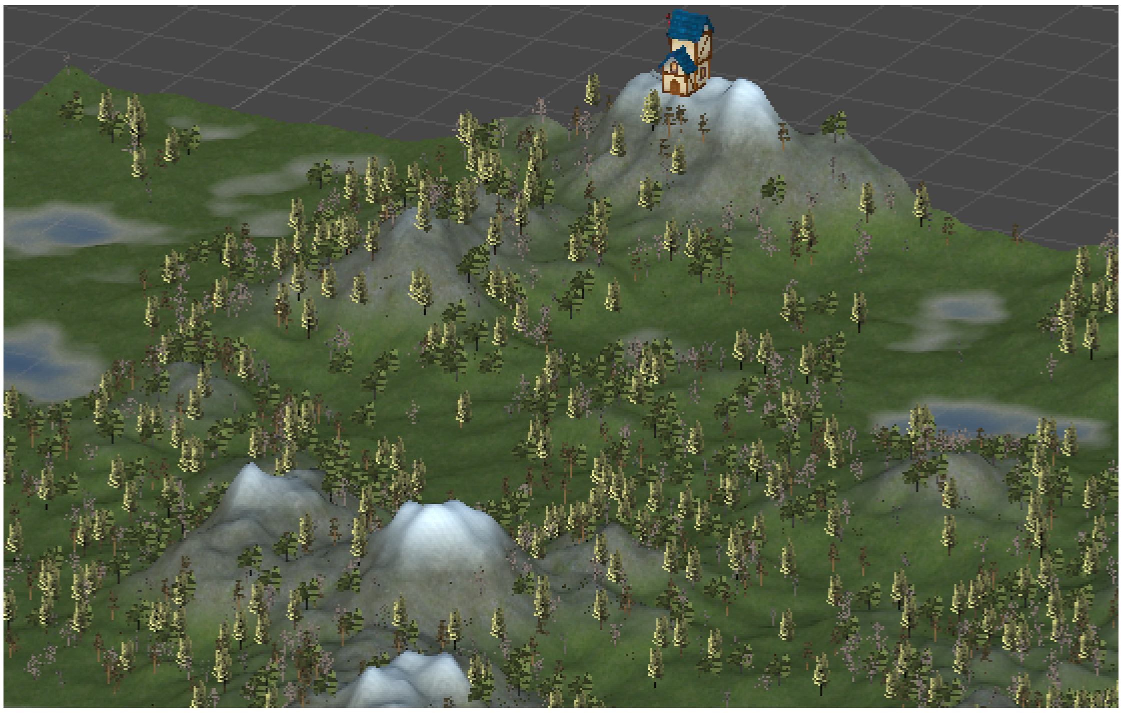


Figure 9. Closeup of a map populated with trees, bushes and a cabin with high detail.

<u>Infinite Terrain:</u>

All the previous figures are generated with the <u>MapPreview</u> object, which allows us to generate samples without having to run the game. Once the game is executed, this will disappear, and another object called <u>MapGenerator</u> will enter in place. The player exists in the scene but it is not instantiated. The MapGenerator will generate several pieces like ones created in the MapPreview around the player. These pieces will have a variable Level of Detail depending on their distance with respect to the player.
The MapGenerator will create a thread for every HeightMap that every piece will need, and also another one for every requested Mesh, that will be dependent on the HeightMap.

As a quick summary and in order of execution is:

1. Calculate how many chunks we will have.
2. Update chunks, that will consists on:

   a. Every chunk will be saved in a dictionary, where the key is the position (x, y). If we already have a chunk there, it will be checked if it needs an update.
   b. If not, a new chunk will be created.
3. Instantiate the player
4. Build the NavMesh around the player.
5. Check the movement of the player and if we need to update the chunks because of the view of the player, apply the chunk update again.

As a result, we can have a map like the one in figure 8, where it will be updated as the player moves.

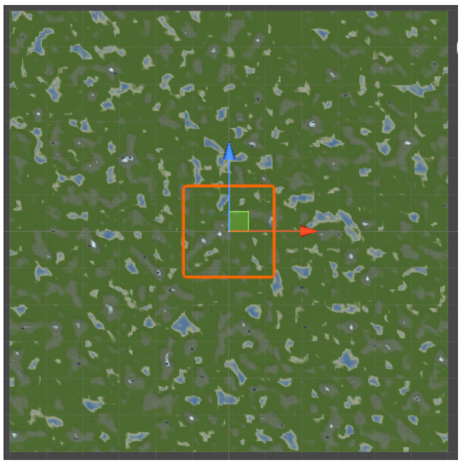
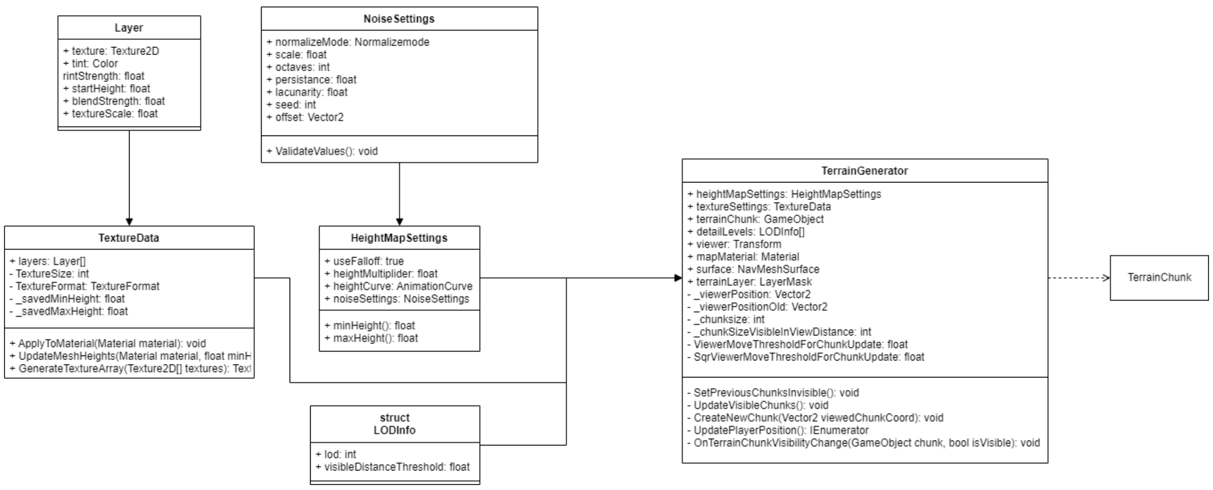


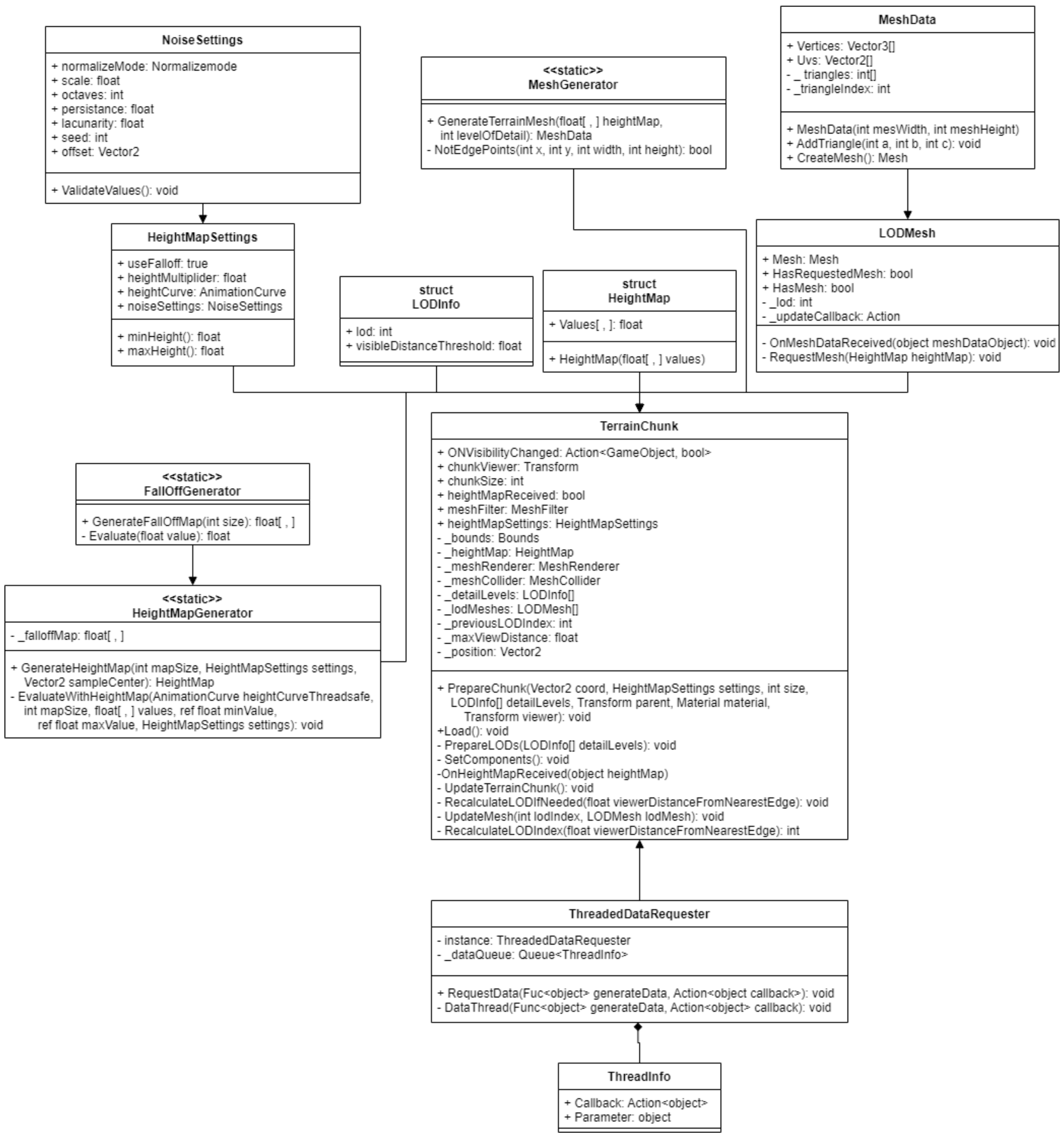
Figure 10. Top-down view of a 5x5 terrain generated by chunks.

The structure of the code regarding the TerrainGenerator class is:


Layer
+ texture: Texture2D
+ tint: Color
rintStrength: float
+ startHeight: float
+ blendStrength: float
+ textureScale: float
NoiseSettings
+ normalizeMode: Normalizemode
+ scale: float
+ octaves: int
+ persistance: float
+ lacunarity: float
+ seed: int
+ offset: Vector2
+ ValidateValues(): void
TextureData
+ layers: Layer[]
- TextureSize: int
- TextureFormat: TextureFormat
- _savedMinHeight: float
- _savedMaxHeight: float
+ ApplyToMaterial(Material material): void
+ UpdateMeshHeights(Material material, float minH
+ GenerateTextureArray(Texture2D[] textures): Text
HeightMapSettings
+ useFalloff: true
+ heightMultiplider: float
+ heightCurve: AnimationCurve
+ noiseSettings: NoiseSettings
+ minHeight(): float
+ maxHeight(): float
struct
LODInfo
+ lod: int
+ visibleDistanceThreshold: float
TerrainGenerator
+ heightMapSettings: HeightMapSettings
+ textureSettings: TextureData
+ terrainChunk: GameObject
+ detailLevels: LODInfo[]
+ viewer: Transform
+ mapMaterial: Material
+ surface: NavMeshSurface
+ terrainLayer: LayerMask
- _viewerPosition: Vector2
- _viewerPositionOld: Vector2
- _chunksize: int
- _chunkSizeVisibleInViewDistance: int
- ViewerMoveThresholdForChunkUpdate: float
- SqrViewerMoveThresholdForChunkUpdate: float
- SetPreviousChunksInvisible(): void
- UpdateVisibleChunks(): void
- CreateNewChunk(Vector2 viewedChunkCoord): void
- UpdatePlayerPosition(): IEnumerator
- OnTerrainChunkVisibilityChange(GameObject chunk, bool isVisible): void
TerrainChunk

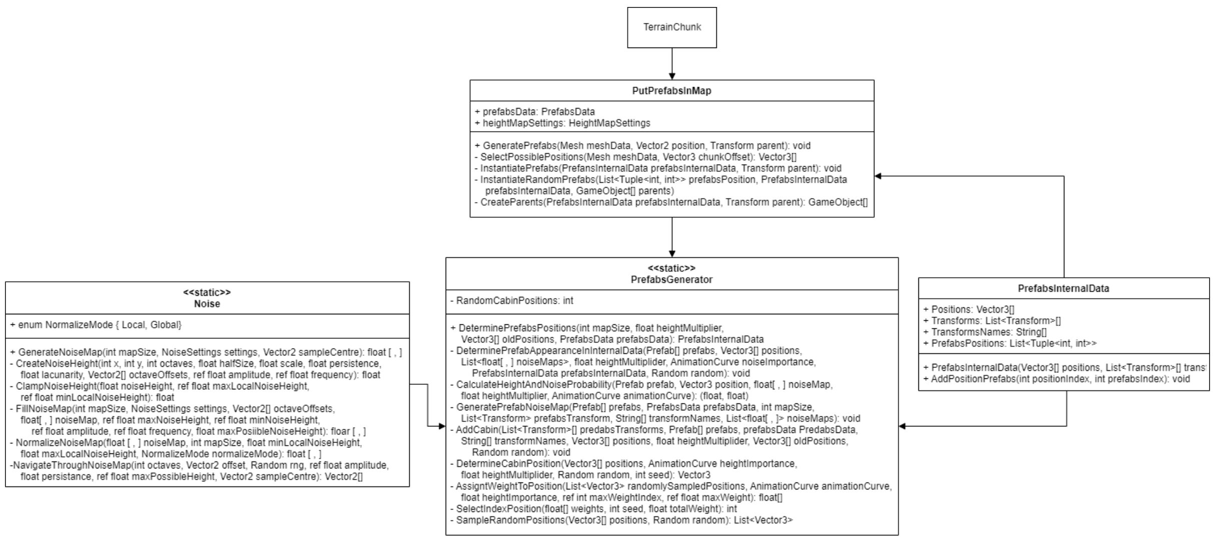
UML 2. Part of TerrainGenerator structure.

On another diagram, we can have a closer look to the TerrainChunk part:


NoiseSettings
+ normalizeMode: Normalizemode
+ scale: float
+ octaves: int
+ persistance: float
+ lacunarity: float
+ seed: int
+ offset: Vector2
+ ValidateValues(): void
<<static>>
MeshGenerator
+ GenerateTerrainMesh(float[,] heightMap, int levelOfDetail): MeshData
- NotEdgePoints(int x, int y, int width, int height): bool
MeshData
+ Vertices: Vector3[]
+ Uvs: Vector2[]
- _ triangles: int[]
- _triangleIndex: int
+ MeshData(int mesWidth, int meshHeight)
+ AddTriangle(int a, int b, int c): void
+ CreateMesh(): Mesh
HeightMapSettings
+ useFalloff: true
+ heightMultiplider: float
+ heightCurve: AnimationCurve
+ noiseSettings: NoiseSettings
+ minHeight(): float
+ maxHeight(): float
struct
LODInfo
+ lod: int
+ visibleDistanceThreshold: float
struct
HeightMap
+ Values[,]: float
+ HeightMap(float[,] values)
LODMesh
+ Mesh: Mesh
+ HasRequestedMesh: bool
+ HasMesh: bool
- _lod: int
- _updateCallback: Action
- OnMeshDataReceived(object meshDataObject): void
- RequestMesh(HeightMap heightMap): void
TerrainChunk
+ ONVisibilityChanged: Action<GameObject, bool>
+ chunkViewer: Transform
+ chunkSize: int
+ heightMapReceived: bool
+ meshFilter: MeshFilter
+ heightMapSettings: HeightMapSettings
- _bounds: Bounds
- _heightMap: HeightMap
- _meshRenderer: MeshRenderer
- _meshCollider: MeshCollider
- _detailLevels: LODInfo[]
- _lodMeshes: LODMesh[]
- _previousLODIndex: int
- _maxViewDistance: float
- _position: Vector2
+ PrepareChunk(Vector2 coord, HeightMapSettings settings, int size, LODInfo[] detailLevels, Transform parent, Material material, Transform viewer): void
+Load(): void
- PrepareLODs(LODInfo[] detailLevels): void
- SetComponents(): void
-OnHeightMapReceived(object heightMap)
- UpdateTerrainChunk(): void
- RecalculateLODIfNeeded(float viewerDistanceFromNearestEdge): void
- UpdateMesh(int lodIndex, LODMesh lodMesh): void
- RecalculateLODIndex(float viewerDistanceFromNearestEdge): int
<<static>>
FallOffGenerator
+ GenerateFallOffMap(int size): float[,]
- Evaluate(float value): float
<<static>>
HeightMapGenerator
- _falloffMap: float[,]
+ GenerateHeightMap(int mapSize, HeightMapSettings settings, Vector2 sampleCenter): HeightMap
- EvaluateWithHeightMap(AnimationCurve heightCurveThreadsafe, int mapSize, float[,] values, ref float minValue, ref float maxValue, HeightMapSettings settings): void
ThreadedDataRequester
- instance: ThreadedDataRequester
- _dataQueue: Queue<ThreadInfo>
+ RequestData(Fuc<object> generateData, Action<object callback>): void
- DataThread(Func<object> generateData, Action<object> callback): void
ThreadInfo
+ Callback: Action<object>
+ Parameter: object

UML 3. Terrain chunk structure.

And finally, the prefab instantiation part:


TerrainChunk
PutPrefabsInMap
+ prefabsData: PrefabsData
+ heightMapSettings: HeightMapSettings
+ GeneratePrefabs(Mesh meshData, Vector2 position, Transform parent): void
- SelectPossiblePositions(Mesh meshData, Vector3 chunkOffset): Vector3[]
- InstantiatePrefabs(PrefansInternalData prefabsInternalData, Transform parent): void
- InstantiateRandomPrefabs(List<Tuple<int, int>> prefabsPosition, PrefabsInternalData prefabsInternalData, GameObject[] parents)
- CreateParents(PrefabsInternalData prefabsInternalData, Transform parent): GameObject[]
<<static>>
Noise
+ enum NormalizeMode { Local, Global}
+ GenerateNoiseMap(int mapSize, NoiseSettings settings, Vector2 sampleCentre): float [,]
- CreateNoiseHeight(int x, int y, int octaves, float halfSize, float scale, float persistence, float lacunarity, Vector2[] octaveOffsets, ref float amplitude, ref float frequency): float
- ClampNoiseHeight(float noiseHeight, ref float maxLocalNoiseHeight, ref float minLocalNoiseHeight): float
- FillNoiseMap(int mapSize, NoiseSettings settings, Vector2[] octaveOffsets, float[,] noiseMap, ref float maxNoiseHeight, ref float minNoiseHeight, ref float amplitude, ref float frequency, float maxPosiibleNoiseHeight): floar [,]
- NormalizeNoiseMap(float [,] noiseMap, int mapSize, float minLocalNoiseHeight, float maxLocalNoiseHeight, NormalizeMode normalizeMode): float [,]
-NavigateThroughNoiseMap(int octaves, Vector2 offset, Random rng, ref float amplitude, float persistance, ref float maxPossibleHeight, Vector2 sampleCentre): Vector2[]
<<static>>
PrefabsGenerator
- RandomCabinPositions: int
+ DeterminePrefabsPositions(int mapSize, float heightMultiplier, Vector3[] oldPositions, PrefabsData prefabsData): PrefabsInternalData
- DeterminePrefabAppearanceInInternalData(Prefab[] prefabs, Vector3[] positions, List<float[,] noiseMaps>, float heightMultiplider, AnimationCurve noiseImportance, PrefabsInternalData prefabsInternalData, Random random): void
- CalculateHeightAndNoiseProbability(Prefab prefab, Vector3 position, float[,] noiseMap, float heightMultiplier, AnimationCurve animationCurve): (float, float)
- GeneratePrefabNoiseMap(Prefab[] prefabs, PrefabsData prefabsData, int mapSize, List<Transform> prefabsTransform, String[] transformNames, List<float[,]> noiseMaps): void
- AddCabin(List<Transform>[] predabsTransforms, Prefab[] prefabs, prefabsData PredabsData, String[] transformNames, Vector3[] positions, float heightMultiplider, Vector3[] oldPositions, Random random): void
- DetermineCabinPosition(Vector3[] positions, AnimationCurve heightImportance, float heightMultiplider, Random random, int seed): Vector3
- AssigntWeightToPosition(List<Vector3> randomlySampledPositions, AnimationCurve animationCurve, float heightImportance, ref int maxWeightIndex, ref float maxWeight): float[]
- SelectIndexPosition(float[] weights, int seed, float totalWeight): int
- SampleRandomPositions(Vector3[] positions, Random random): List<Vector3>
PrefabsInternalData
+ Positions: Vector3[]
+ Transforms: List<Transform>[]
+ TransformsNames: String[]
+ PrefabsPositions: List<Tuple<int, int>>
+ PrefabsInternalData(Vector3[] positions, List<Transform>[] trans
+ AddPositionPrefabs(int positionIndex, int prefabsIndex): void

UML 4. Prefab instantiation structure

<u>Characters (Player):</u>

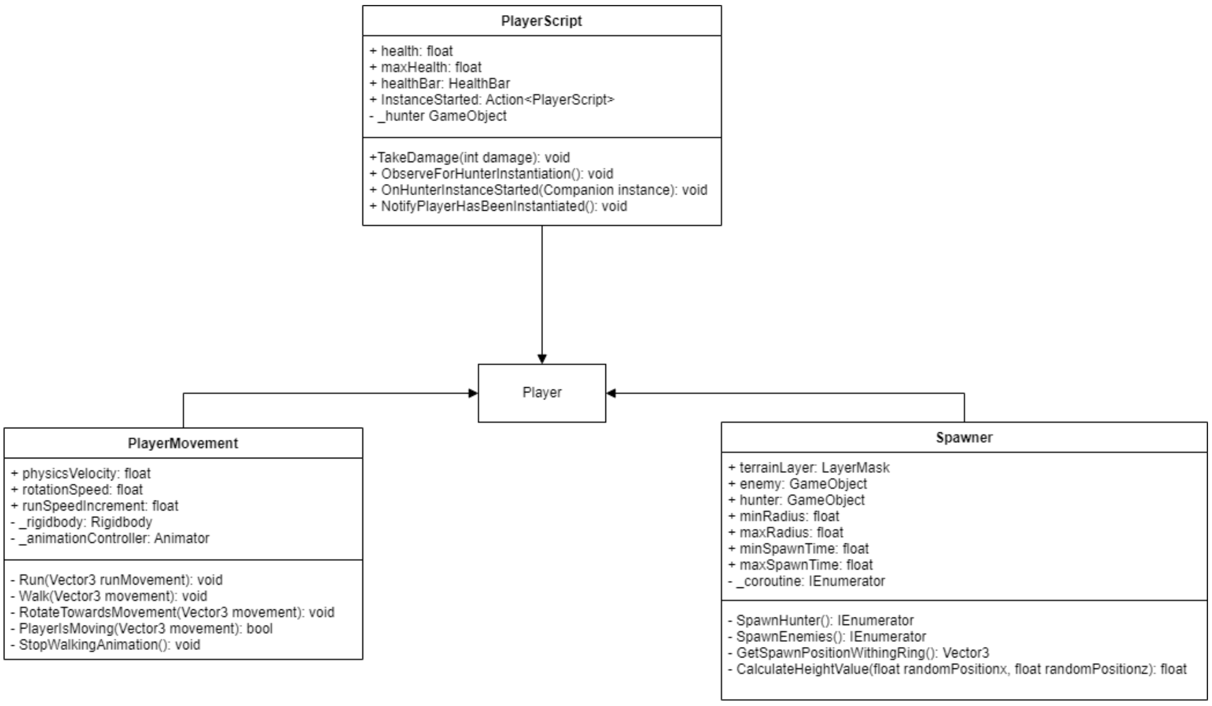
The player is controlled with simple WASD movements and the camera is automatically located in one position. The player is fairly simple, as it just has movement with a simple sprint as SHIFT. It has a life bar but as mentioned before the player will not die, this was just to test the damages of the enemy attack. It will also observe the Hunter with an observer pattern to check when it is instantiated, so it will get a reference of it automatically instead of checking every update of the game.

Apart from movement and being protected from the Hunter, it has two other functionalities:

1. Pick up arrows for the hunter
2. Spawn enemies

The spawning of the enemies is a functionality that is attached to the player for commodity reasons. The enemies will spawn within a ring shaped form around the player, so they are not created neither too far, nor on top of the player. When each enemy is killed, it will spawn an arrow in the floor that the player can pick up and will be added to the hunter's quiver, as he has a limited amount of arrows.

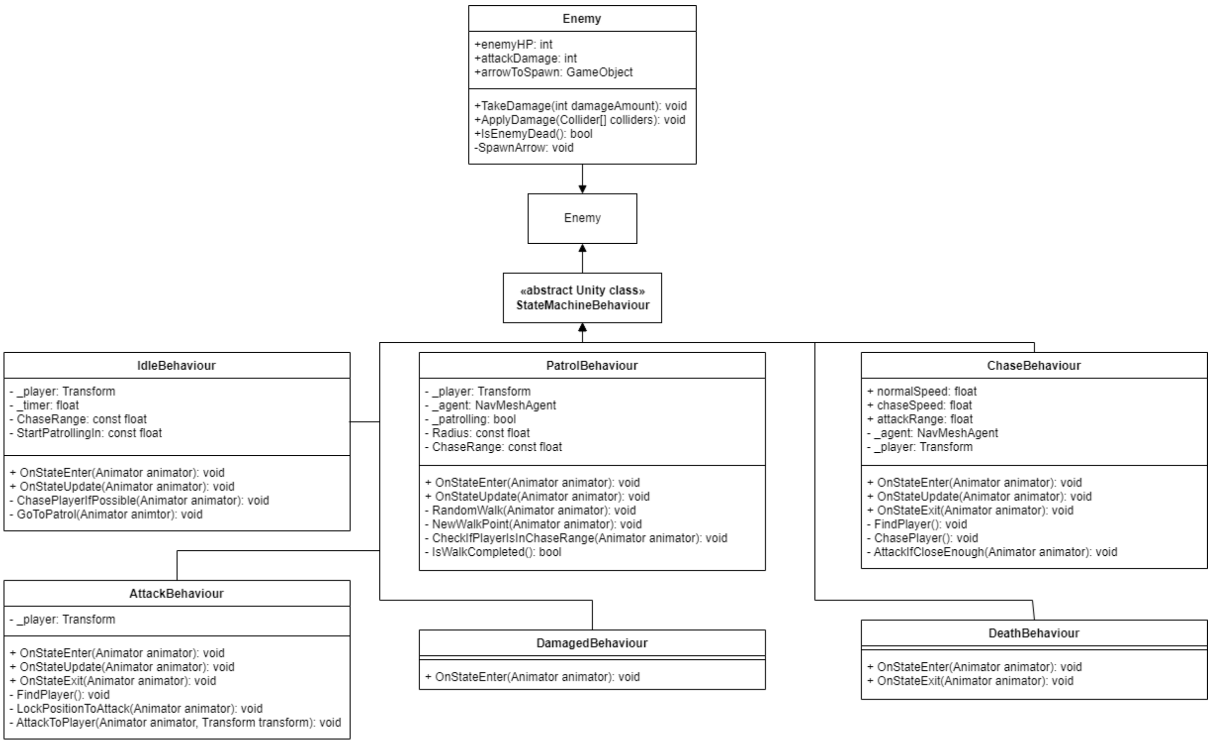
A UML diagram of the Player can be seen here:


PlayerScript
+ health: float
+ maxHealth: float
+ healthBar: HealthBar
+ InstanceStarted: Action<PlayerScript>
- _hunter GameObject
+TakeDamage(int damage): void
+ ObserveForHunterInstantiation(): void
+ OnHunterInstanceStarted(Companion instance): void
+ NotifyPlayerHasBeenInstantiated(): void
Player
PlayerMovement
+ physicsVelocity: float
+ rotationSpeed: float
+ runSpeedIncrement: float
- _rigidbody: Rigidbody
- _animationController: Animator
- Run(Vector3 runMovement): void
- Walk(Vector3 movement): void
- RotateTowardsMovement(Vector3 movement): void
- PlayerIsMoving(Vector3 movement): bool
- StopWalkingAnimation(): void
Spawner
+ terrainLayer: LayerMask
+ enemy: GameObject
+ hunter: GameObject
+ minRadius: float
+ maxRadius: float
+ minSpawnTime: float
+ maxSpawnTime: float
- _coroutine: IEnumerator
- SpawnHunter(): IEnumerator
- SpawnEnemies(): IEnumerator
- GetSpawnPositionWithingRing(): Vector3
- CalculateHeightValue(float randomPositionx, float randomPositionz): float

UML 5. Player structure.

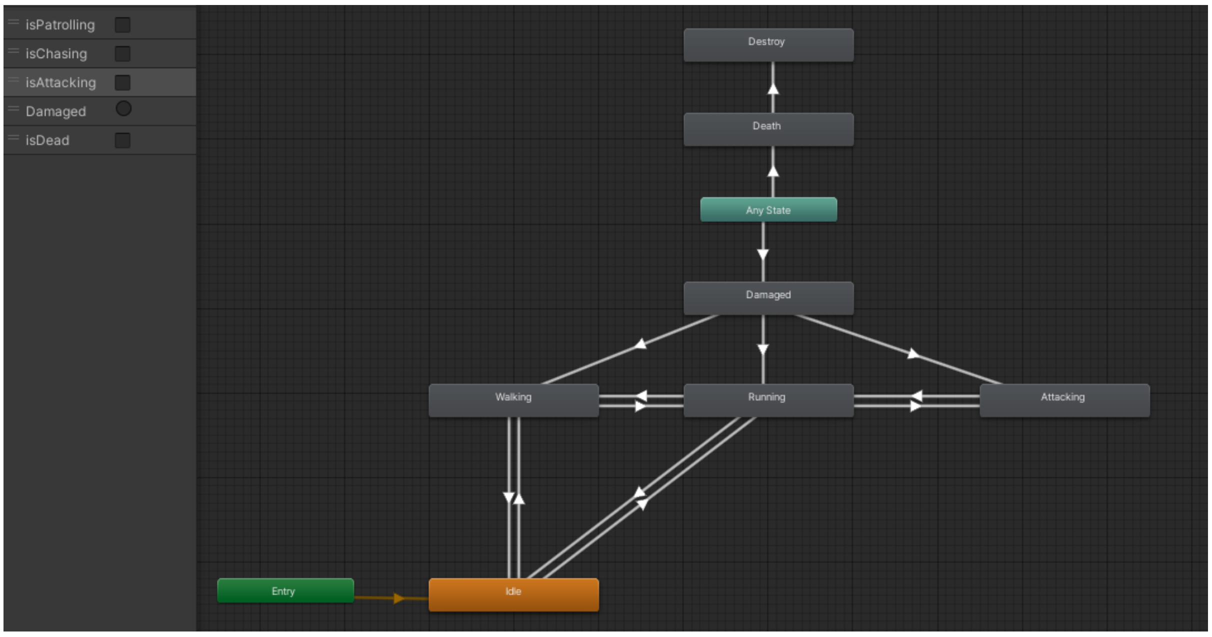
<u>Characters (Enemy):</u>

The enemy behaviour is implemented using the functionalities that Unity brings us with the animator system, where each animation can also correspond to a state, making an easy mix between animations and behaviours into the same Finite State Machine. The states of the enemy controller can be seen here:


Enemy
+enemyHP: int
+attackDamage: int
+arrowToSpawn: GameObject
+TakeDamage(int damageAmount): void
+ApplyDamage(Collider[] colliders): void
+IsEnemyDead(): bool
-SpawnArrow: void
Enemy
«abstract Unity class»
StateMachineBehaviour
IdleBehaviour
- _player: Transform
- _timer: float
- ChaseRange: const float
- StartPatrollingIn: const float
+ OnStateEnter(Animator animator): void
+ OnStateUpdate(Animator animator): void
- ChasePlayerIfPossible(Animator animator): void
- GoToPatrol(Animator animtor): void
PatrolBehaviour
- _player: Transform
- _agent: NavMeshAgent
- _patrolling: bool
- Radius: const float
- ChaseRange: const float
+ OnStateEnter(Animator animator): void
+ OnStateUpdate(Animator animator): void
- RandomWalk(Animator animator): void
- NewWalkPoint(Animator animator): void
- CheckIfPlayerIsInChaseRange(Animator animator): void
- IsWalkCompleted(): bool
ChaseBehaviour
+ normalSpeed: float
+ chaseSpeed: float
+ attackRange: float
- _agent: NavMeshAgent
- _player: Transform
+ OnStateEnter(Animator animator): void
+ OnStateUpdate(Animator animator): void
+ OnStateExit(Animator animator): void
- FindPlayer(): void
- ChasePlayer(): void
- AttackIfCloseEnough(Animator animator): void
AttackBehaviour
- _player: Transform
+ OnStateEnter(Animator animator): void
+ OnStateUpdate(Animator animator): void
+ OnStateExit(Animator animator): void
- FindPlayer(): void
- LockPositionToAttack(Animator animator): void
- AttackToPlayer(Animator animator, Transform transform): void
DamagedBehaviour
+ OnStateEnter(Animator animator): void
DeathBehaviour
+ OnStateEnter(Animator animator): void
+ OnStateExit(Animator animator): void

UML 6. Enemy Structure

Using the Unity controller, can be seen as:



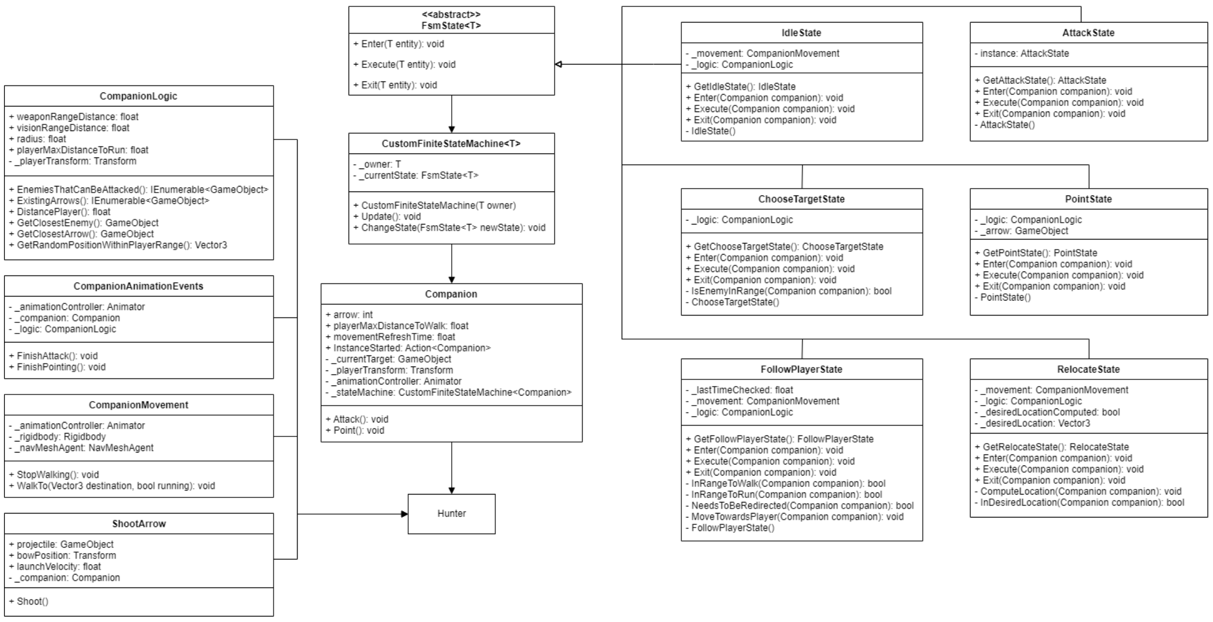
Animation Controller 1. Enemy animation controller.

It can be explained as:

1. The enemy enters to idle behaviour as default. After some time, it starts to patrol/walk changing its state.
2. If the player is in range, it will change its state to chase/running.
3. When the enemy reaches the player, it will do an attack animation changing the state again.
4. If it is not in attack animation, it will keep chasing.
5. If it is not in chase range, it will start to patrol again.
6. If it is damaged by the hunter, it will change the state and will do the damage animation.
7. If at any point the life of the enemy goes below zero, it changes the state to death, plays the death animation, disappears and spawns an arrow.

<u>Characters (Hunter):</u>

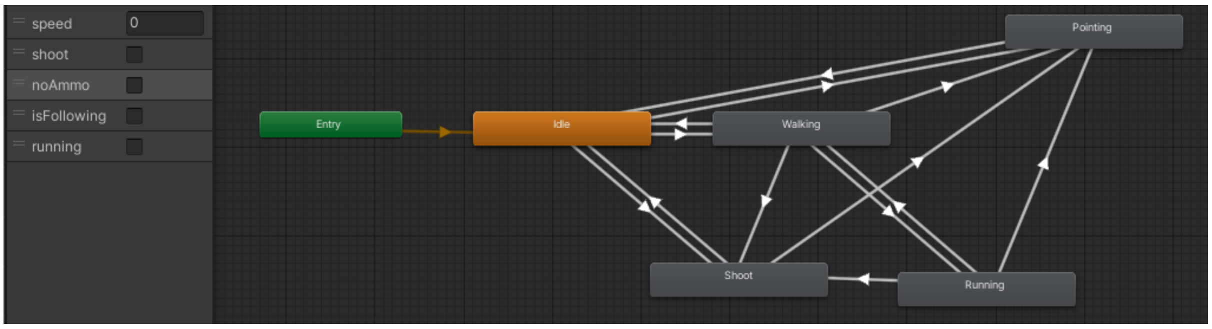
In order to do different things, the FSM used in the Hunter is hand made and it is not used in the animation sistem directly as the Enemy. A visual explanation of the Hunter can be seen here:



<<abstract>> FsmState<T>
+ Enter(T entity): void
+ Execute(T entity): void
+ Exit(T entity): void
CustomFiniteStateMachine<T>
- _owner: T
- _currentState: FsmState<T>
+ CustomFiniteStateMachine(T owner)
+ Update(): void
+ ChangeState(FsmState<T> newState): void
Companion
+ arrow: int
+ playerMaxDistanceToWalk: float
+ movementRefreshTime: float
+ InstanceStarted: Action<Companion>
- _currentTarget: GameObject
- _playerTransform: Transform
- _animationController: Animator
- _stateMachine: CustomFiniteStateMachine<Companion>
+ Attack(): void
+ Point(): void
Hunter
CompanionLogic
+ weaponRangeDistance: float
+ visionRangeDistance: float
+ radius: float
+ playerMaxDistanceToRun: float
- _playerTransform: Transform
+ EnemiesThatCanBeAttacked(): IEnumerable<GameObject>
+ ExistingArrows(): IEnumerable<GameObject>
+ DistancePlayer(): float
+ GetClosestEnemy(): GameObject
+ GetClosestArrow(): GameObject
+ GetRandomPositionWithinPlayerRange(): Vector3
CompanionAnimationEvents
- _animationController: Animator
- _companion: Companion
- _logic: CompanionLogic
+ FinishAttack(): void
+ FinishPointing(): void
CompanionMovement
- _animationController: Animator
- _rigidbody: Rigidbody
- _navMeshAgent: NavMeshAgent
+ StopWalking(): void
+ WalkTo(Vector3 destination, bool running): void
ShootArrow
+ projectile: GameObject
+ bowPosition: Transform
+ launchVelocity: float
- _companion: Companion
+ Shoot()
IdleState
- _movement: CompanionMovement
- _logic: CompanionLogic
+ GetIdleState(): IdleState
+ Enter(Companion companion): void
+ Execute(Companion companion): void
+ Exit(Companion companion): void
- IdleState()
AttackState
- instance: AttackState
+ GetAttackState(): AttackState
+ Enter(Companion companion): void
+ Execute(Companion companion): void
+ Exit(Companion companion): void
- AttackState()
ChooseTargetState
- _logic: CompanionLogic
+ GetChooseTargetState(): ChooseTargetState
+ Enter(Companion companion): void
+ Execute(Companion companion): void
+ Exit(Companion companion): void
- IsEnemyInRange(Companion companion): bool
- ChooseTargetState()
PointState
- _logic: CompanionLogic
- _arrow: GameObject
+ GetPointState(): PointState
+ Enter(Companion companion): void
+ Execute(Companion companion): void
+ Exit(Companion companion): void
- PointState()
FollowPlayerState
- _lastTimeChecked: float
- _movement: CompanionMovement
- _logic: CompanionLogic
+ GetFollowPlayerState(): FollowPlayerState
+ Enter(Companion companion): void
+ Execute(Companion companion): void
+ Exit(Companion companion): void
- InRangeToWalk(Companion companion): bool
- InRangeToRun(Companion companion): bool
- NeedsToBeRedirected(Companion companion): bool
- MoveTowardsPlayer(Companion companion): void
- FollowPlayerState()
RelocateState
- _movement: CompanionMovement
- _logic: CompanionLogic
- _desiredLocationComputed: bool
- _desiredLocation: Vector3
+ GetRelocateState(): RelocateState
+ Enter(Companion companion): void
+ Execute(Companion companion): void
+ Exit(Companion companion): void
- ComputeLocation(Companion companion): void
- InDesiredLocation(Companion companion): bool

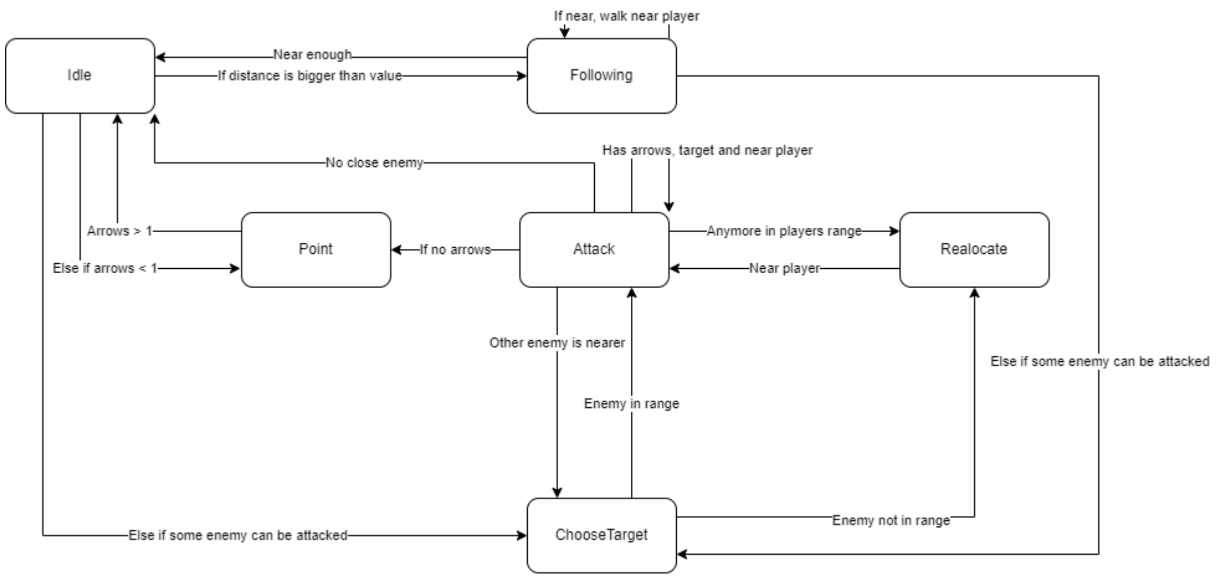
UML 7. Companion structure

It also has 6 states, like the enemy, but the behaviour is a bit more complex. Using the animation system of Unity can help us to have a better overview. But as we mentioned, it is not directly related to the controller as the enemy behaviours.



Animation Controller 2. Hunter animation controller.

As is not as “simple” as the enemy, it can be better seen in this diagram.



FSM 1. Hunter FMS.

Summary:

Apart from this, there are other implementations that are not explained here. For example the shooting attack from the Hunter or the Attack itself from the enemy. Also, the available nav mesh is not the entire terrain, it is a square centered around the player with a fixed size that is updated every certain time as he moves.

To sum up, the parts that were used in this project are:

- Procedural generation
- Balance of charge
- Finite States Machine
- Navigation Agents
- Meshes and textures
- Animation and animation events
- Editor functionalities

Among others such as the control of physics, code flow, etc.
Needless to say that the code could be much less coupled and more organized, and also the project structure. Other parts, like prefab instantiation could also be improved, since instantiation with full detailed mesh is a little expensive as it is done right now with brute force.

Project Link: https://github.com/AlbertoSoutullo/AIUnityTesting
Explanatory Video: https://youtu.be/LQyubv3AlMs
Explanatory Video Part 2: https://youtu.be/CoICF2Dp1GM
Inspired by Sebastian Lague incredible tutorials.

Thank you for your time,
Alberto