Unity Game 2D Documentation

Jagoda Janowska

https://github.com/AventixQ/game_2D_unity

Contents

1. Basic information about game construction.

1. Game

The player's goal is to reach the red orb while collecting as many coins as possible and avoiding enemy attacks. The player has to navigate through a labyrinth containing items such as coins, hearts, golden hearts, and enemies. If the player gets lost in the labyrinth, they can return to the beginning of the game with the help of a blue orb. Every 30 seconds in 25 different locations, a new enemy appears, so the faster the player finds the path to the red orb, the fewer enemies they will encounter.

2. The map consists of several layers of objects:

- User Interface

- Blocking

- Consumables

- Enemies

The user interface is where information about the player's health level and the items they have in their inventory is stored.

The blocking layer is the layer on which the character moves. This layer interacts with all the other layers.

The consumables layer is where all "edible" items like gold and health are placed.

The enemies layer is where the enemies are stored.

3. Used scripts

During the game, scripts of various types are used: [Note: This part of the text seems to be cut off. If you have more content to add, please provide it.]**Class MonoBehaviour** is a base class that many Unity scripts derive from. MonoBehaviour offers life cycle functions that make it easier to develop with Unity.

- Inventory.cs

- Slot.cs

- HealthBar.cs

- SpawnPoint.cs

- MovementController.cs

- Consumable.cs

- Character.cs

- Player.cs

- Enemy.cs

- Wander.cs

- RPGCameraManager.cs

RPGGameManager.cs

Class CinemachineExtension is a base class for a Cinemachine Virtual Camera extension module.

- RoundCameraPos.cs

ScriptableObject is a data container that you can use to save large amounts of data, independent of class instances.

- HitPoints.cs

- Item.cs

4. Created Objects

In addition to Unity scripts, the creation of objects is possible. Objects in the game have an appropriate script embedded in them, enabling their functionality. There are four types of objects in the game:

- Coin

- Heart

- GoldenHeart

- HitPoints

Among these, the first three objects contain the "Item.cs" script, which defines:

- Object name

- Object animations

- Quantity

- Whether the object can be stacked

- Object type (coin / health)

The last object is created using the "HitPoints" script and contains a numerical "value."

5. Prefabricats

Unity's Prefab system allows you to create, configure, and store a GameObject complete with all its components, property values, and child GameObjects as a reusable Asset. The Prefab Asset acts as a template from which you can create new Prefab instances in the Scene.

In the game we will use:

- CoinObject

- HeartObject

- GoldenHeartObject

- EnemyObject

- PlayerObject

- SpawnPoint

- Slot

- InventoryObject

- HealthBarObject

- LevelMove

- LevelStay

- RPGCameraManager

- RPGGameManager

The Start() and Update() methods are included in the scripts by default. The former starts at the very beginning of the game, while playing the latter is triggered every time you refresh the game.

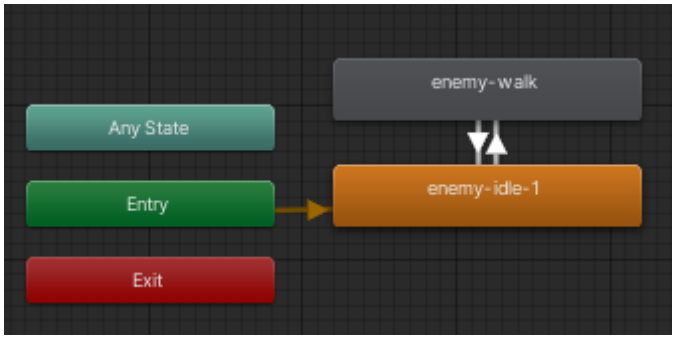
6. Graphics

We will use publicly available projects from the Unity Asset Store for the graphics of our game – map, enemies, items and player.

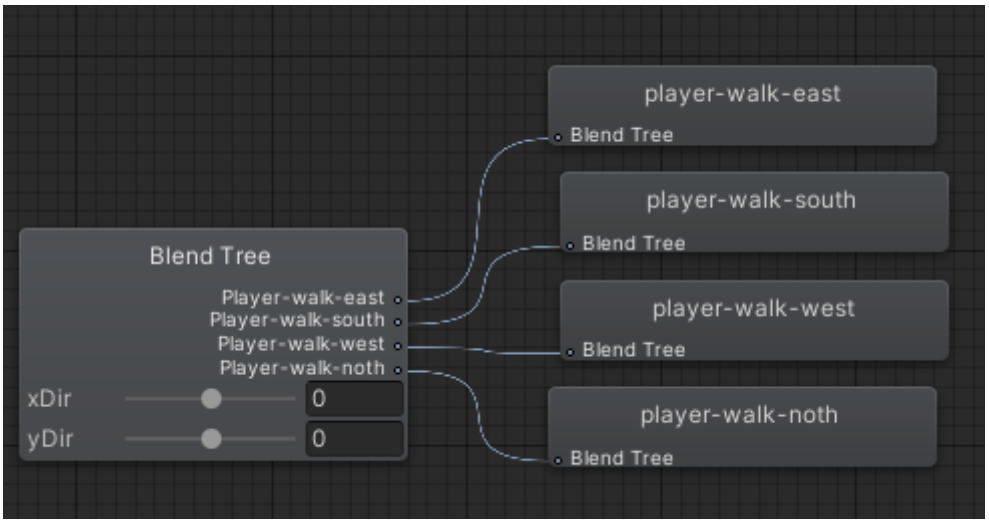
2. Game elements

1. Character Animations

To set characters in motion, you need to create character animations and controllers. All of this is done within the Unity environment without the need for programming. The environment will automatically generate animations from the images we provide. It is within our scope to arrange animations based on user actions. For this purpose, correct arrangement and transitions of animations in the controller are necessary.



1 – Animation of movement for Enemy



2 - Animation of movement for Player using Blend Tree

2. Character Collisions with Objects

The programming of character collisions can also be added within Unity itself. This is achieved using the "Collider" component in its various forms. To enable collisions between the player and objects in the game world, the player must be given this component. Everything must be tailored to the appropriate layers in the program.

We also want to assign physical attributes to the player, for which the Rigidbody2D object can be used.

3. Character Movement Control

Character movement control is programmed in the **MovementControls.cs** script. Part of this code (connected with configuration of Blend Tree) was taken from the internet.

In the script, the movement speed is defined, which can be manually set within Unity. The variable "movement" is also declared as:

Vector2 movement = new Vector2();

Vector2 is a built-in class that holds two-dimensional vectors or points. This variable is used to store the position of player and enemy objects.

We also declare the Rigidbody2D component. We do this to obtain a reference to the same component assigned to the PlayerObject. It is used for moving the character.

In the script, we utilize various methods, including:

- FixedUpdate() - it is called by the Unity engine at regular time intervals. Inside it, the MoveCharacter() method is called.

- MoveCharacter() - the code responsible for moving the character, which returns a value of 1, 0, or -1 based on the pressed WSAD key. The character moves along one axis according to the provided information, multiplied by an appropriate movement speed.

- UpdateState() - the code sends information to the state animator about the character's intended position. It has been specifically tailored for animations created as a Blend Tree.

4. Camera manager

To make the camera follow the hero, lock when the map ends, and respawn with the player, the Cinemachine component was utilized. This is an operator that controls the camera's position and lens settings. It can be assigned a target to follow, designate a path to move along, and switch between different paths. All this data can be implemented in Unity without the need for programming. However, Collider should be employed, and map boundaries should be created as a new layer to prevent the player from venturing beyond.

To stabilize the camera, an appropriate script needs to be written, in my case: **RoundCameraPos.cs**.

In this script, we no longer use the Monobehavior class, but instead, CinemachineExtension. We set the definition of PPU (Pixels Per Unit) to 32 pixels within it.

We also override the PostPipelineStageCallBack() method. This method is required in all classes derived from CinemachineExtension. It's invoked by the camera after calculating the bounds.

In the mentioned method, we use the Round() method, which rounds the given value. This allows us to position the camera with pixel accuracy.

5. Character Class

The Character class is a class from which both the Player and Enemy classes inherit. It contains methods and properties that are common to all characters in the game. The script implementing this class is saved in Character.cs.

The "abstract" keyword indicates that instances of this class cannot be created on their own. Instead, derived classes should be created.

The properties maxHitPoints and StartingHitPoints are responsible for the current and maximum health of the character, respectively.

The KillCharacter() method is marked as "virtual," indicating that it can be overridden if the current implementation proves insufficient. The code implements a basic character kill, but additional functionality related to this operation will be needed in derived classes.

Similarly, two methods, ResetCharacter() and DamageCharacter(), are created using the "abstract" keyword. The former resets the character object to its initial state for reuse. The latter is called by another character object to weaken the current character. The arguments include the amount of health points to be deducted and an interval if the character needs to be weakened at regular time intervals.

FlickerCharacter() is used to show that a character has been hit and has lost some of its health. Coroutine is used for this purpose. The SpriteRenderer component, assigned to our characters for proper image display, is utilized. We change the component's color to red, simulating damage to the character in the game. The coroutine is paused for 0.1 seconds, after which the character's color is changed back to white.

6. Player Class

The Player class is responsible for all actions related to our player character. In addition to the attributes inherited from the Character class, our player has features such as a health bar and an inventory. Its script is saved under the name **Player.cs**.

The OnEnable() method invokes the ResetCharacter() function. This function will be called every time the object is enabled and activated. A similar method is present in the Enemy class.

ResetCharacter() is an overridden function from the same-named function in the Character class. In this case, it resets both the inventory and the health bar. This is achieved by assigning existing variables to inventoryPrefab and healthBarPrefab, effectively "resetting" the properties of both items.

If we want a certain event to act as a trigger, all we need to do is mark the appropriate option in Unity. However, defining what should happen when the trigger is activated is a different story. Such actions are defined in the functions OnTriggerEnter2D() and OnTriggerExit2D(). The former is used in the script.

OnTriggerEnter2D(Collider2D collision) checks if the player has encountered any trigger objects. These triggers include objects with the tag "CanBePickedUp," such as coins and hearts. The method determines that if we enter such an object and are able to pick it up, the shouldDisappear variable is updated accordingly and the object is irreversibly destroyed outside the switch. If a collision is detected but the object has different properties or cannot be picked up (for example, the player has full health HP: 100 and cannot increase it further), the variable should remain false, leaving the object untouched.

In the above function, for HEALTH-type objects, we call the AdjustHitPoints() function. It's responsible for adding an appropriate amount of health to the health bar and displaying the amount of health in the console. This function also uses a limiter that prevents the heart from being collected if it exceeds the character's health limit.

DamageCharacter() is a coroutine that weakens the player at regular intervals. The coroutine uses the yield return new WaitForSeconds() statement to pause the code execution for a specific time. As long as the player takes damage, they will flicker using the FlickerCharacter() function from the Character class. Additionally, health points will be subtracted. If the player's health is insufficient, they are killed. Otherwise, the loop pauses for the number of seconds specified in the interval argument and then resumes. The loop only breaks if the character dies.

KillCharacter() first calls base.KillCharacter(), which is the KillCharacter() method of the base class. This method removes the GameObject object representing the player character. Only the inventory and health bar need to be destroyed, which is done using Destroy().

7. Enemy class

The Enemy class is very similar to the Player class with a few exceptions. First of all, we don't need to worry about inventory and health bars for enemies, as they don't possess them. The enemy appears at its spawn point at regular intervals (in this case, every 30 seconds), so this needs to be considered in the program. The script handling the enemy mechanics is located in Enemy.cs.

Starting from the beginning, the strength with which the enemy inflicts damage to the player is determined. A property is also assigned to allow us to stop the DamageCharacter() coroutine.

Similar to the player, OnEnable() method calls the ResetCharacter() method.

Due to the changes described above, the ResetCharacter() function will no longer concern itself with inventory and health bars. Instead, it will simply store the initial hitPoints value in the hitPoints variable.

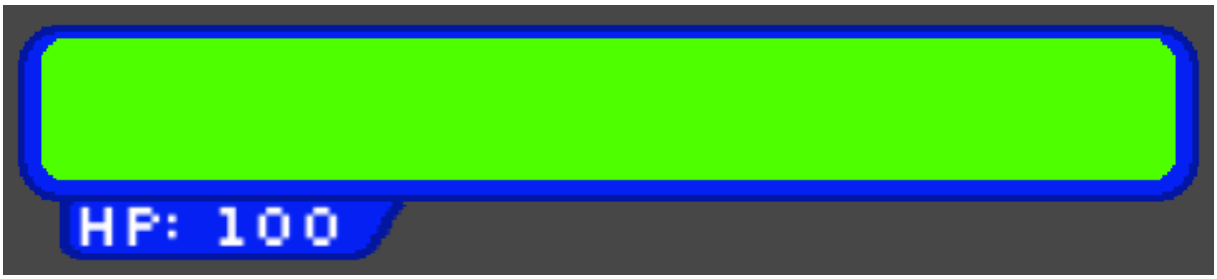
This time, we won't use the OnTriggerEnter() function because the enemy is the trigger, not the player. Instead, we want to detect collisions using the OnCollisionEnter2D(Collision2D collision) function. If the enemy collides with an object tagged as "Player," we identify the specific player. Then, we check if the DamageCharacter() coroutine of the player object is currently running. If it's not, we initiate it. In the coroutine's arguments, we pass the number of health points and the interval, as the enemy should weaken the player throughout the collision between the two characters.

OnCollisionExit2D() is invoked when the current and other colliders objects stop colliding with each other. The DamageCharacter() coroutine is stopped and damageCoroutine is set to null.

In this class, DamageCharacter() works exactly the same way as in the Player class. As soon as the health stored in the hitPoints variable drops to a certain amount, the enemy is killed.

8. Health Bar

The health bar is a UI Canvas object that Unity automatically inserts into a dedicated screen. The bar itself consists of several objects merged into a whole, including several graphic layers.



3 – Player Health Bar

In addition to the graphical aspect, it is controlled by two scripts. One of them is **HealthBar.cs**, and the other is **HitPoints.cs**.

HitPoints is a ScriptableObject and, within its class, it only has a float-type value.

On the other hand, HealthBar belongs to the MonoBehaviour class and contains information about the character's health amount, images for display, text, and maximum health. It doesn't have any methods other than the standard ones, Start() and Update().

At the start of the game, the hero's maximum health is saved. A safeguard has been added (as there were many issues with this part of the code), which checks whether a character reference is assigned to the HealthBar.

In the Update method, the health bar's fill and the underlying text are constantly monitored and adjusted.

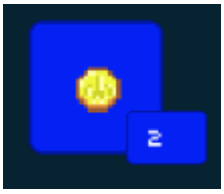
The Health Bar is heavily used in the Character, Player, and Enemy classes, as you've seen in the previous sub-headings.

9. Inventory

An inventory has also been created for the player, i.e. a place where the things collected by him are stored.

This time, the inventory wasn't manually set up in Unity, as was the case with the Health Bar, but it appears automatically with a predetermined number of slots through the code. We use the Slot prefab for its operation, which handles the functioning of individual slots, as well as the InventoryObject prefab, which brings all the slots together. A specific script was created for each of them: **Inventory.cs** and **Slot.cs**.

A **Slot** consists of a background, a count of the collected items of a specific type, and an image. In the Slot.cs script, a text has been programmed to appear. The exact place where subsequent slots appear can be set in the Unity editor itself by clicking on our predicate, and then on its properties.



4 - Slot with coins inside

The **Inventory.cs** script aims to manage the player's inventory and the appearance of the bar. The first thing we'll implement in it is a reference to the Slot prefab. Next, we determine the number of available slots – in my case, there will be 5 slots. The next elements are arrays: an array to store images, an array to store references to objects of type Item, and an array to store text.

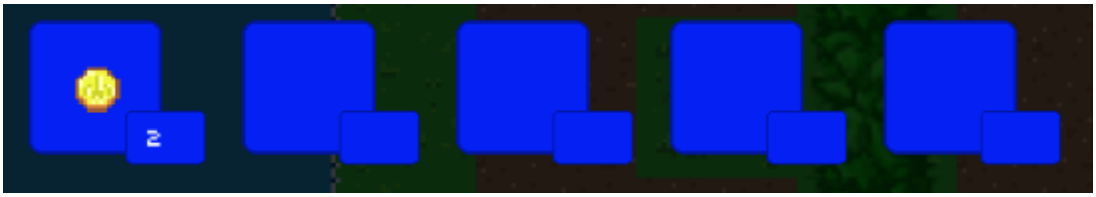
The CreateSlots() function will be called at the start. Its goal is to dynamically create instances of the Slot prefab. Within a loop, it iterates the specified number of times to create new slots, giving them names, adding them to the slots array, and adding their images to the itemImages array.

In addition to the code that creates our slots, we need to create a function that adds items to them. This is where AddItem() comes into play. For each slot in the inventory, we check whether the item we want to add is already placed there. If:

a) YES – we increase the item's quantity and update the displayed text.

b) NO – we add the item to an empty slot, set the quantity to 1, and add the image.

This algorithm creates a simplified player inventory.



5 - Player inventory

3. Singleton

Singleton is used when only one instance of a specific class is needed throughout the entire application's runtime. Singleton is useful in situations where a single class contains functionalities utilized by other classes, such as in the case of a game manager coordinating the game flow. It also serves as a public, official access point to the class and its functionalities. Lazy instantiation is commonly used, where the instance is created only when it's required.

This section of code was copied form book and adapted to fit my game concept.

1. RPG Game Manager

The RPGGameManager object with the **RPGGameManager.cs** script is utilized as a singleton because the game's progression is orchestrated by only one class at a time.

In the script, the static property sharedInstance is used to reference the singleton object.

In the Awake() method, we check whether the sharedInstance property has been initialized and if its value is different from the current instance of the class. If the sharedInstance property is initialized and points to a different instance than the current one, the new instance is destroyed. There can only be one instance of the RPGGameManager class. If there's only one instance, it is assigned to the sharedInstance property.

The entire code that prepares the scene is placed in the Start() method. Inside it, there's the SetupScene() method, and within that, there's the SpawnPlayer() method.

SpawnPlayer() checks whether a SpawnPoint has been placed in the game. If so, it spawns the player and positions the camera on them. More about spawning will be explained in the upcoming chapters.

The Update() function allows the game to be exited using the ESC key.

2. RPG Camera Manager

To make the camera follow the character as it moves around the map, the RPGCameraManager class was created, and it's an object with its **RPGCameraManager.cs** script.

Once again, a singleton was used to create the object. The CinemachineVirtualCamera property, written in the script, holds a reference to the virtual camera. We will programmatically set this reference, so we want to hide it in the Unity program interface.

The Awake() method, as in the previous script, implements the singleton pattern. Then, the virtual camera is searched for in the scene using the "VirtualCamera" tag, and the virtual camera's properties are stored in the virtualCamera variable.

In Unity, you can set various properties of the camera, including labels, "follow" options, and camera size.

3. Other game elements

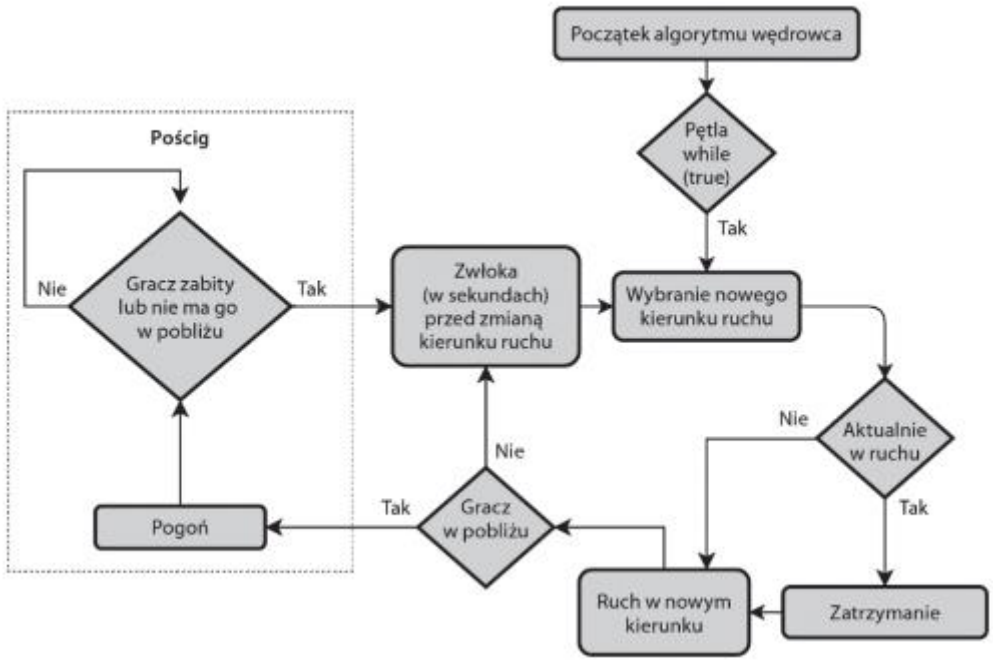
1. Respawn Points

Characters in the game, both the player and their enemies, respawn at specific locations in the scene at regular intervals. To achieve this functionality, a SpawnPoint prefab is created, and it's assigned a script that handles character respawning. This prefab is used for both respawning the player and the enemies.

The **SpawnPoint.cs** script requires information about the prefab to be used, either PlayerObject or EnemyObject in our case, and the time after which the character should respawn again. For the player, the time is set to 0 because we don't want to "multiply" them. For the monsters in my game, the respawn time has been set to 30 seconds, and this SpawnPoint prefab is used a total of 25 times.

2. Wanderer algorithm

The wanderer's algorithm was created based on diagram found on the Internet. It is saves as **Wander.cs**.



6 - Basic idea of wanderer algorithm (in Polish)

Based on this scheme, I've created a program that incorporates elements such as:

- Three properties describing the enemy's movement speed. The first determines the speed during pursuit, the second is the normal speed, and the third is the current speed - one of the two mentioned above.

- The directionChangeInterval property is defined using the Unity editor. It represents the delay before the enemy changes its movement direction.

- The FollowPlayer boolean determines whether the enemy should follow the player.

- The moveCoroutine holds a reference to the coroutine controlling character movements. In the case of the enemy, it slightly moves them in each frame in the specified direction.

- Properties storing references to the RigidBody2D and Animator components of the GameObject.

- The targetTransform property will hold a reference to the Transform component of the player character for pursuit.

- A property determining the direction of character movement.

- The current angle of character movement.

The Start() method allows us to initially obtain references to the Animator, RigidBody2D, and CircleCollider components and store them in memory. It records the current speed, calls the WanderRoutine() coroutine, which is the beginning of the wanderer algorithm.

Next, the WanderRoutine() loop was created, which, as per the blueprint, is an infinite loop. In this method, the enemy wanders indefinitely. The ChooseNewEndpoint() method changes the angle and position towards which the Enemy is heading. It is updated initially and every time the player enters the enemy's range. Then, using a conditional statement, it is checked whether the enemy is currently moving. If so, it needs to be stopped before redirecting it in a new direction. This stops the current coroutine and calls the Move() coroutine. Finally, the coroutine's execution is paused, then the loop resumes, and a new endpoint is chosen.

The function ChooseNewEndpoint() has already been mentioned, but within it, another function called Vector3FromAngle is also executed. This method converts an angle in degrees to radians and returns a Vector3 result.

The Move() coroutine is responsible for moving the character with a given speed from the current position to a point defined by the endPosition property. The method uses the Rigidbody2D component for this purpose. To achieve this, it checks the remaining distance to the point. If the distance is greater than float.Epsilon, practically 0, the pursuit is triggered.

When the enemy is chasing the player, the targetTransform property holds a reference to the Transform component of the player object, rather than being null. In this case, the endPosition property will be assigned the player's position from the targetTransform property. As a result, the enemy will start moving towards the player character rather than the initial endPosition.

The Move() coroutine uses the Rigidbody2D component of the enemy to move it. Before executing further operations, a check for a reference to this component is performed. The boolean parameter "isWalking" is set to true, which triggers a walking animation. This variable is initially set in Unity during animation creation. This initiates the transition of the enemy to the "enemy-walk-1" state and plays the corresponding animation.

The Vector3.MoveTowards() method calculates the displacement used in the Rigidbody2D component. The MovePosition() method moves the enemy character to the position calculated in the previous step and stored in the newPosition variable. The remaining distance is updated using the sqrMagnitude property. The coroutine is then yielded. After the loop, the "isWalking" variable is set to false.

To actually chase the enemy, the player entering the enemy's field of view needs to be targeted. This can be accomplished using the OnTriggerEnter() method, as mentioned before. If an object enters the collision field:

a) Check if the object is the player – it has the "Player" tag and the enemy's followPlayer property is set to true.

b) The current speed value changes to pursuitSpeed.

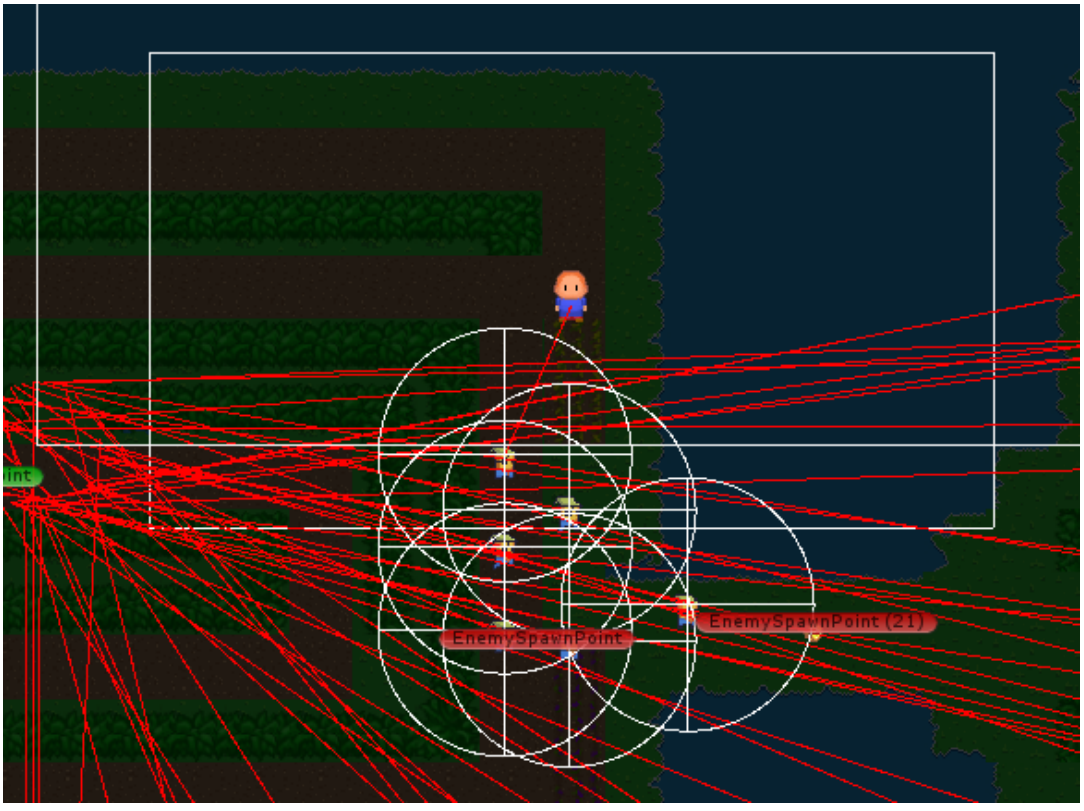
c) A reference to the player's Transform object is stored in targetTransform.

d) If the enemy is currently moving, it needs to be stopped using StopCoroutine.

e) The Move() coroutine is called.

The same principle applies to OnTriggerExit(). If the pursuitSpeed property in the Enemy class is less than the movementSpeed property in the Player class, the player can escape from the enemy. When the player exits the range, this method is called. Consequently, when the enemy loses sight of the player, it starts wandering aimlessly again. The method is very similar to OnTriggerEnter(). Once again, it checks if the object exiting the collision is the player. Then the enemy loses orientation and stops. This is done by setting the isWalking variable to false. The currentSpeed once again assumes the value of wanderSpeed, the Move() coroutine is interrupted, and the targetTransform becomes null.

In addition, auxiliary effects have been added to the code, such as OnDrawGizmos() and Update(), where lines are drawn to represent the enemy's field of view and the direction they are currently moving. Both functions are displayed in Gizmo mode during game runtime but are not accessible to the user.



7 - Additional effectsw

3. Teleportation

The goal of the game is to navigate through the maze and reach the red orb, which teleports us to the next level. If we get lost on the map, we also have the option to use the blue orbs placed around to return to the starting point. Therefore, the game's mechanics revolve around transitioning to the appropriate scenes as soon as our player collides with the corresponding objects.

8 - Level move orb

In the script **LevelMove.cs**, we provide the index of the scene we want to transition to. Currently, the game doesn't have a second map, so it either moves to the scene with the "You win!" message for index 1, or teleports back to the player's spawn point for index 0. In the OnTriggerEnter2D() function, it's checked whether the object entering it is the player, and then the scene corresponding to the provided index is loaded.

9 - Screen after winning game

4. Sources

https://learn.unity.com/
https://assetstore.unity.com/
https://www.youtube.com/watch?v=mpM0C6quQjs
Game Development with Unity for .NET Developers Book