Software Engineering
Senior Project

Towin Beast The Game
Technical Design

Group Members

Doğa Ünal
Bora Armağan Koyuncu
Sinan Küçükyılmaz

Table of Contents

1. Executive Summary

Towin Beast is a 2D platformer game with action components developed for Windows desktop platform. The game allows players to experience fast-paced and challenging gameplay which also supports gamepad controllers. Pixel graphics art syle also used in this game which creates fun and engaging environment. Current demo version of the project consists of three levels which are Tutorial, The Space Station and Boss Level. In these levels players will be play as a space soldier called Towin who tries to stop his evil-self who tries to conquer all dimensions.

2. Background and Context

2.1 Purpose

The purpose of this document is to outline the technical design and architecture of the solution , including the technology and infrastructure.

2.2 Intended Audience

The intended audience of this document are jury of Izmir University of Economics senior project evaluation commission.

2.3 Objectives

The VMAS demonstration website was designed with the following core objectives in mind:

- Provide a flexible environment for demonstrating the use of VMAS;
- Demonstrate the integration of other DSE and third party web services, namely Image Web Service (IWS) and Google Maps;
- Provide authentication to secure the site and provide to access live and test VMAS environments;
- Provide information for prospective users of the VMAS service;
- Provide a framework for the future development of other VMAS client applications;

2.4 Design Goals

The following design goals were taken into account:

- Game to be developed based on OOP best practices, including the use of recommended naming conventions, software design patterns and optimization practices of gaming industry;
- Code to be clear with sufficient comments and documentation;
- New levels and in game items must be easily added and configurable without requiring heavy code changes;
- Object creation and destruction will be memory friendly;
- Rendering will be easily handled even by low-standard desktop computers;

2.5 Assumptions and Constraints

- Developers have familiarity with Microsoft .Net, Visual Studio 2017 , Unity Game Engine;
- Game to be compatible with Microsoft Windows 7 and newer versions.

3. Technology and Infrastructure

3.1 Application Technologies

The technologies used to develop and deploy this system are:

- Microsoft .Net Framework version 4.5
- Unity Game Engine version 2017.3
- Mono Develop

The languages used are:

- C#

The development environment is:

- Visual Studio 2017 Community

3.2 Hardware Requirements

The demonstration website has been deliberately kept lightweight. Although the hardware requirements will depend upon factors such as the expected load, concurrent users, and frequency and duration of visits, it is anticipated that a current model entry level server with the following specification would be sufficient for this system:

- 2 GHZ CPU
- 1 GB Ram
- 500 MB Available HDD Space

3.3 Software Requirements

Target computer should be working on Microsoft Windows 7 and new versions. Microsoft .Net Framework version 4.5 is also required on the computer.

4. Solution Architecture

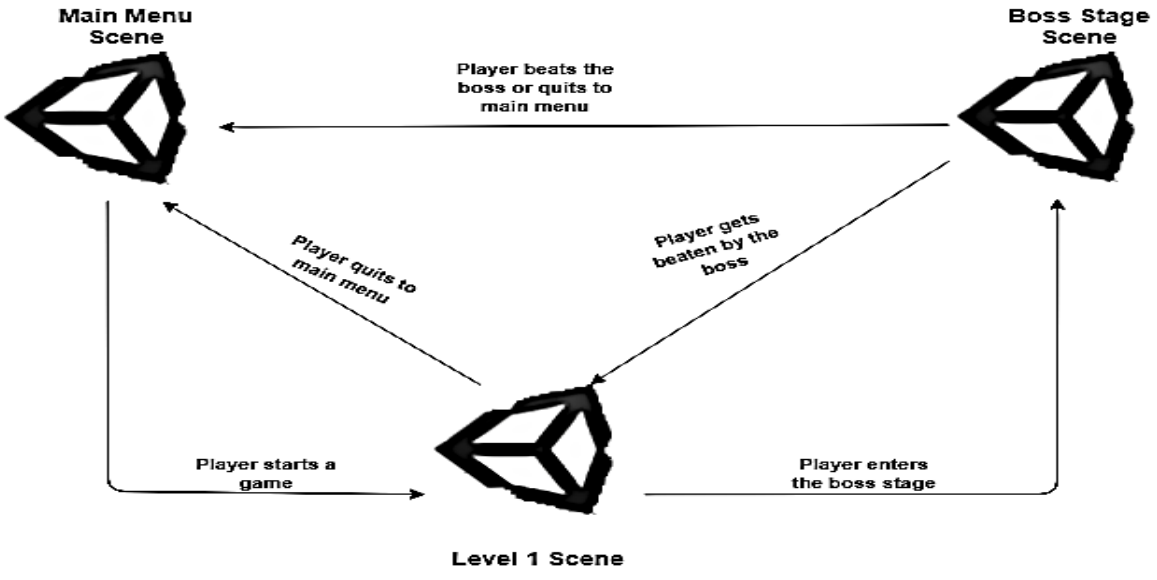
Unity Scene Interaction

In Unity Engine environments and menus are contained by unique files which are called “Scenes”. Scenes can be thought as unique levels or set of interfaces. In Towin Beast Demo, there are three scenes which are :

• **Main Menu Scene** : This scene contains starting menu, save profiles and basic options for the game.

• **Level 1 Scene** : This scene is the first level of the game where player starts his/her story and it is the environment where the main game loop exists. All game objects related with game mechanics and design is in this scene such as fighting mechanics, 2D physics controllers, input handlers, enemy mechanics and other items in game environment.

• **Boss Stage Scene** : This scene contains the first boss of the game it uses same game objects, mechanics and technical base with Level 1 Scene.



.4.1 Main Menu Scene

Main Menu Scene is the scene where players starts the game. This scene consists of GUI elements which make player navigate to Level 1 Scene. To get the Level 1 Scene, players either starts a new game or load their previous save file from Save Slots Menu panel. In Save Slots Menu panel there are there save slots. If players want to start a new game, they just simply select an empty save slot. Then a virtual keyboard appears to make players able to type their slot name. After desired slot name defined then a new slot saved to file system as a new save file. If there is no empty slots for players to start a new game, players can overwrite an existing save slot by simply selecting and accepting “overwrite warning” and typing their new slot profile name.

Players are also able to change simple settings and display keyboard/gamepad controller map. Options Menu panel enables players to set soundtrack and sound effects levels. Also players can able to switch between pre-defined keyboard/gamepad key control map profiles.

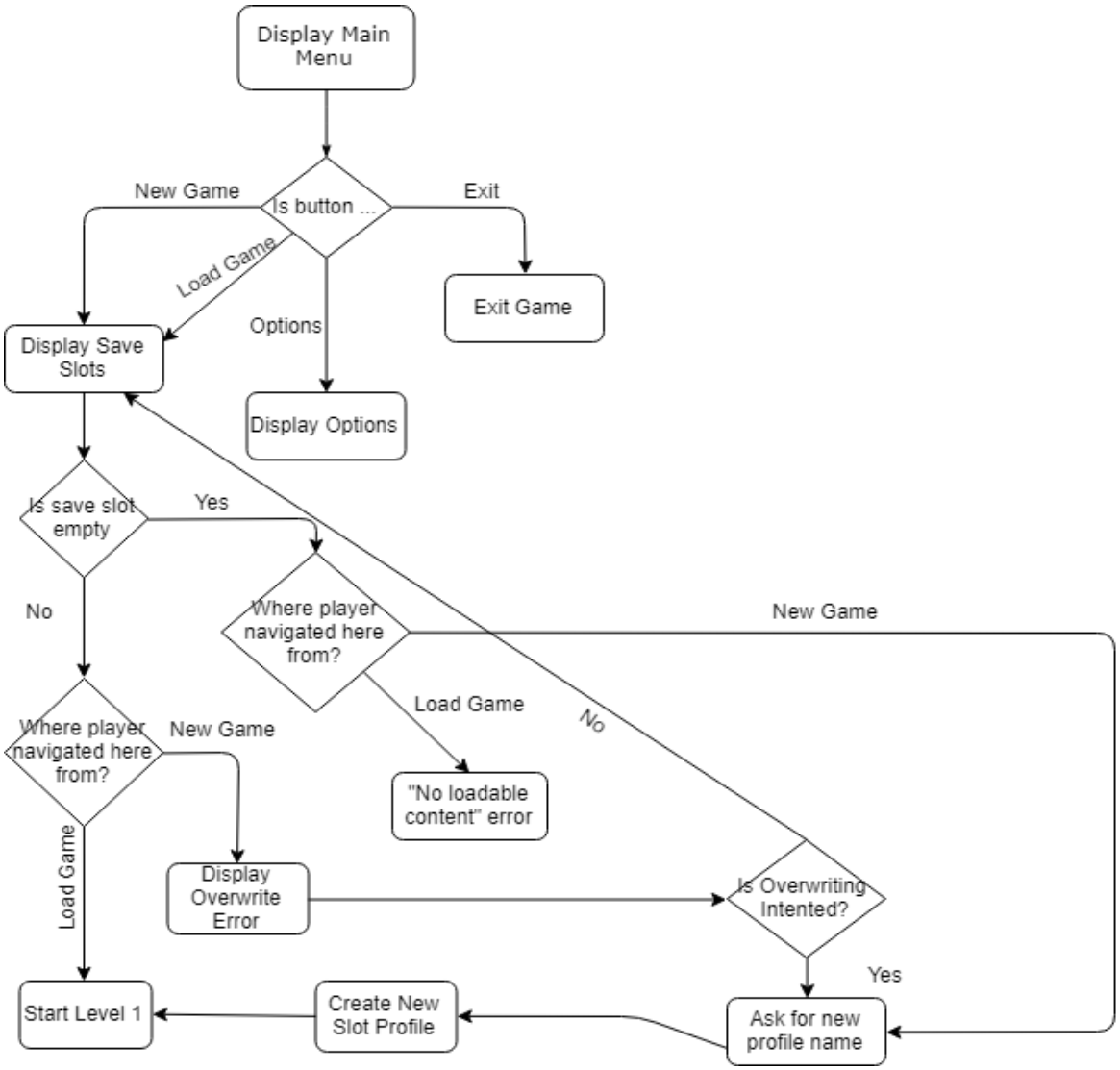
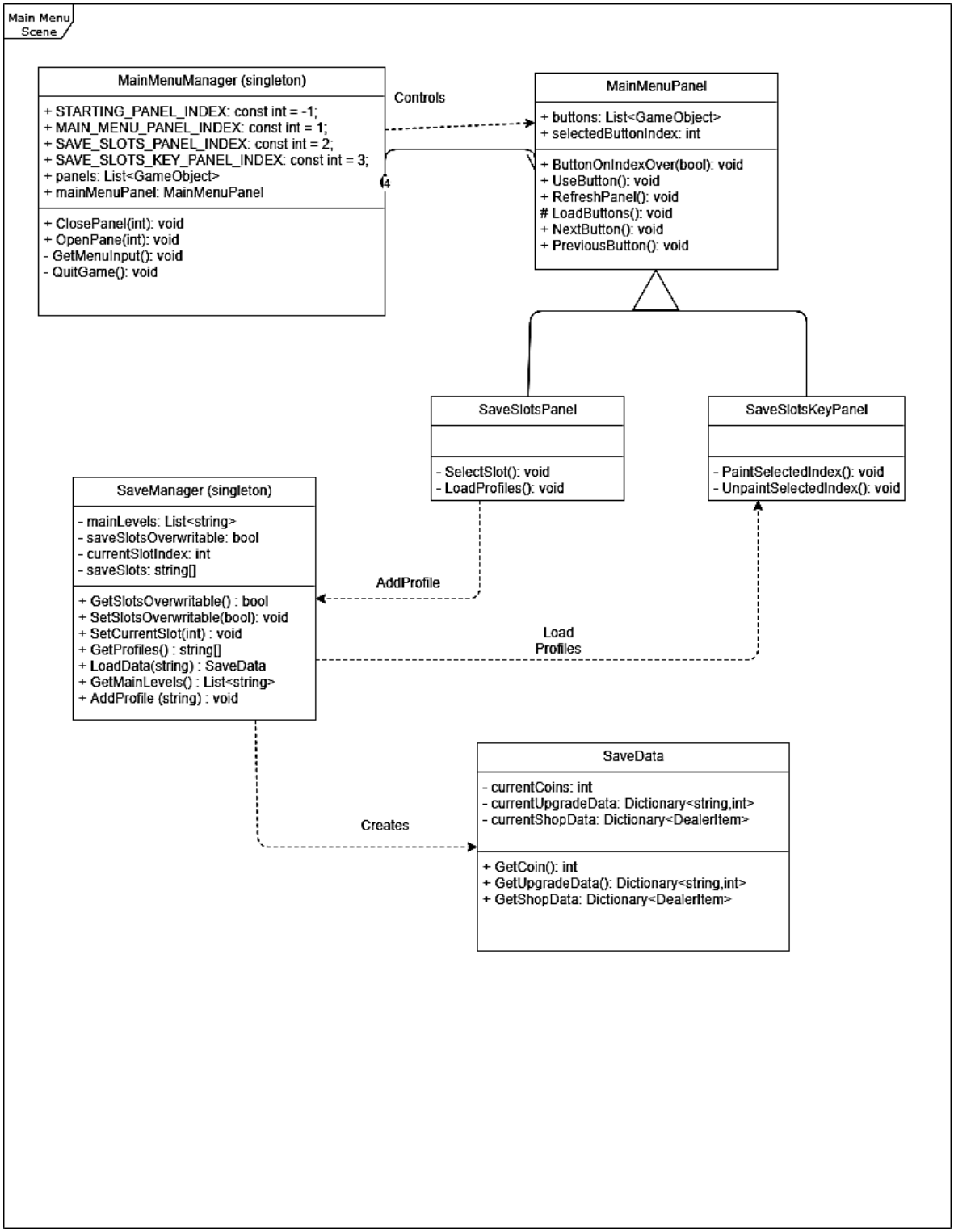


Fig.1: *Main Menu Scene Flowchart*

4.1.1 Main Menu Scene Classes

Main Menu Scene contains following classes in game object hierarchy :

- MainMenuManager
- SaveManager
- MainMenuPanel
- SaveSlotsPanel
- SaveSlotsKeyPanel
- SaveData



4.1.2 Main Menu Manager Class

MainMenuManager class is a singleton class which controls all panels (Save Slots Panel, Main Menu Panel, Options Panel, Virtual Keyboard Panel) and process player inputs to navigate through these panels. When Main Menu Scene started, in Start() method, loads game objects whose type is MainMenuPanel to 'panels' list. Everypanel has constant index in MMM class so that OpenPanel(int) or ClosePanel(int) can be used on desired panel. Because everypanel is inherited from MainMenuPanel, MMM class has a field type of MainMenuPanel which holds the reference of current MainMenuPanel instance. When any panel is used they are upcasted to MainMenuPanel class.

```
void ClosePanel(int panelIndex)
{
    if(panelIndex != -1)
    {
        MainMenuPanelScript heritCaster = (MainMenuPanelScript) panels [panelIndex - 1].GetComponent(typeof(MainMenuPanelScript));
        panels[panelIndex-1].SetActive (false);
        currentPanelBehavior = null;
    }
    else
    {
        Debug.LogError("Invalid panel index.");
        return;
    }

}
```

4.1.3 Main Menu Panel Class

MainMenuPanel is the class controls first panel in the scene and also a base class form SaveSlotsPanel and SaveSlotsKeyPanel. All panels in the scene contains buttons to make players able to navigate between menus so MainMenuPanel class has a game object list called 'buttons' which holds button objects and an integer value called 'selectedButtonIndex' holds index of current button. This 'buttons' list also inherited to child classes and almost all operations in MainMenuPanel are about these button objects.

- LoadButtons() Method :

This method loads child button objects of 'mainMenuPanel' object to 'buttons' list when Start() method invoked. The method also adds Action Listeners to button objects.

- ButtonOnIndexOver(bool isForward) Method:

This method ables user to traverse between button objects in 'buttons' list with keyboard or gamepad inputs which are triggered by GetMenuInput() method in MainMenuManager class. Boolean parameter in the method defines if players traverse forward or backward. Traversing updates current selected button index field and also paints button to red to make players aware of which button their cursor on.

- UseButton() Method:

This method invokes the method which is bound to current selected button object. It simply clicks the selected button.

- RefreshPanel() Method:

When this method called, selected button index gets value of -1 which mean all buttons are unselected. As mentioned, color of selected button becomes red so an unselected button must be its default color so that this method also paint the buttons into their default color.

- NextButton() Method:

This is the method which ables players to navigate to next button in the panel. It simply increases the selected button index count if current index is not its max levels. When NextButton method called, it also invokes ButtonIndexOver(bool) method with boolean value of 'true' so when NextButton method called, it unpaints the previous button and paints the current one. Player button cursor simply iterates forward.

- PreviousButton() Method:

This is the method which ables players to naviate to previous button in the panel. It simply decreases the selected button index count if current index is not its max levels. When PreviousButton method called, it also invokes ButtonIndexOver(bool) method with boolean value of ‘false’ so when PreviousButton method called, it unpaints the next button and paints the current one. Player button cursor simply iterates backward.

4.1.4 Save Slots Panel Class

Save Slots Panel class controls ‘saveSlotsPanel’ object in Main Menu scene. When players wants to start a new game or load their previous games they are navigated to this panel. This panel is a child of Main Menu Panel class so most methods it contains from its parent class. They also have additional class which triggers Save Manager class to manage saving and loading operations.

- SelectSlot() Method:

This method is invoked inside of override version of parent UseButton() method. First, the method checks if players navigated here from New Game or Load Game button from Main Menu Panel. It matters because if players are navigated from New Game button, they are able to create a new profile or overwrite current slots in order to create a new profile. Otherwise, they would not be able to write to slots. To check this, saveSlotsWritable which is a boolean varialbe of Save Manager class. If slots are avaiable to write on them, current slot index is set and players are navigated to Save Slots Key Panel.

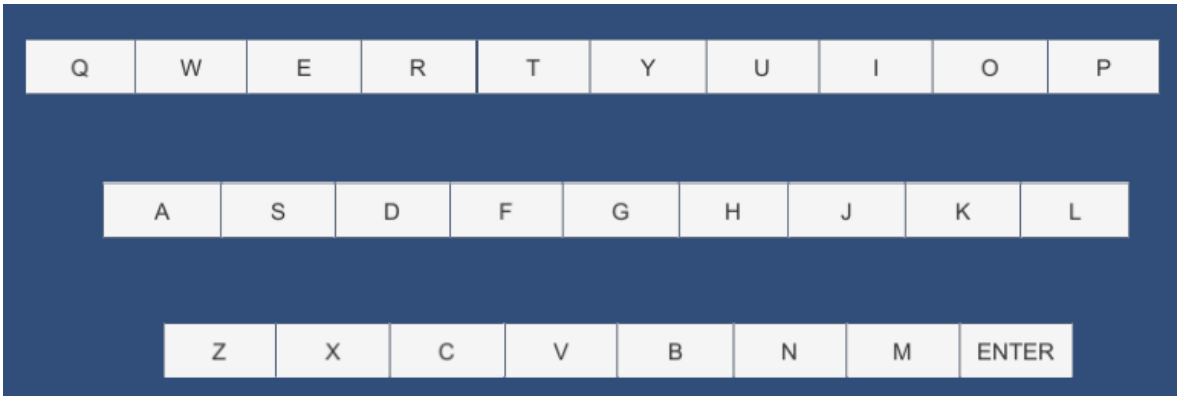
If slots are not writable it means players are navigated to this panel from Laod Game button. In this condition if they choose a button. It invokes LoadData(string) in SaveManager with profile name of current selected button.

- LoadProfiles () Method:

If there are previously saved profiles, SaveSlotsPanel class gets profile names of previous savings from SaveManagerClass then write these names into save slot buttons.

4.1.5 Save Slots Key Panel Class

Save Slots Key Panel class controls ‘saveSlotsVirtualKeyboardPanel’ object in Main Menu scene. When players selects an empty slots to create a new profile, they are navigated to menu of this class. This menu contains of virtual buttons that enables players to type any word to name the selected empty slot.



Because SaveSlotsKeyPanel class is a child of MainMenuPanel class, it has protected LoadButton() method. This method is overridden because buttons in this panel represent a virtual keyboard and it has buttons which correspond to alphabetic letters.

This class also has a field of GameObject to display current profile name which is typed by players. When a button is pressed, a new character from the virtual keyboard concatenated to current text of this GameObjct.

When playes are done with naming, they press ‘ENTER’ button, then they get started to navigate to Level 1 Scene.

- Mark Buttons() Method:

This method sets Text components of loaded buttons as alphabetical characters.

```
string[] letters = {"Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P"
    , "A", "S", "D", "F", "G", "H", "J", "K", "L", "Z", "X", "C", "V", "B", "N", "M","ENTER"
};
for(int i = 0 ; i < buttons.Count ; ++i)
{
    buttons[i].transform.GetComponentInChildren<Text> ().text = letters [i];
}
```

- ToRightButton() Method:

Because players can iterate right,left,up and down on virtual keyboard, this additional classes are required.

ToRightButton method calls NextButton of the parent class because NextButton simply increases the current button index it just iterates left to right over ‘buttons’ list.

- ToLeftButton() Method:

ToLeftButton method calls PreviousButton of the parent class because PreviousButton simply decreases the current button index it just iterates right to left over ‘buttons’ list.

- Overriden Methods NextButton() and PreviousButton():

Overriden NextButton and PreviousButton is changed to iterate up and down over virtual keyboard. According to number of buttons in a row in the virtual keyboard, method defines a index which means ‘up’ and ‘down’. The virtual keyboard cursor simply jumps to that index.

- AddLetter() Method:

This method concetenates a character to the display text object as mentioned in the description. The method passed in the listeners in ‘buttonsList’ when LoadButtons method is called.

- SendProfileToSaveManager() Method:

This method is called when player hit ‘ENTER’ button on the virtual keyboard. It takes Text component of the name display object then invokes AddProfile(string) method with the text. Then loads Level 1 Scene.

4.1.6 Save Manager Class

SaveManager class is a singleton class which is responsible from loading and saving operations of game profiles.

Fields

- string[] mainLevels :

  This array holds names of the levels. According to save data of players, SaveManager choose the correct level name to continue.

- bool saveSlotsWritable:

  This boolean variable holds the value of if a new game profile could be created. It becomes ‘true’ if players navigated here from ‘New Game’ button in the main menu.

- int currentSlotIndex :

  This variable holds the index of which slot is being modified by the player.

- string[] saveSlots:

This array holds names of the save profiles. They are set to default values of ‘Empty Slot’ but when players create a new slot with their name it is updated.

Methods

- bool GetSlotsIfWritable() :

  This method returns value of private saveSlotsWritable variable.

- void SetSlotsWritable(bool):

  This method sets private saveSlotsWritable variable with desired boolean value.

- void SetCurrentSlotIndex(int):

  This method sets private currentSlotIndex variable with desired int value.

- string[] GetProfiles():

  This method returns content of ‘saveSlots’ array which is needed to display names in SaveSlotsPanel class.

- SaveData LoadData(string) :

  According to given profile name, this method finds previous save files from built in PlayerPrefs class of unity and create a new SaveData instance. Then returns it.

- List<string> GetMainLevels():

  Returns content of ‘mainLevels’ array as an accessor.

- void AddProfile(string):

  Adds a new profile name to the current index of ‘saveSlots’ array.

- Void SaveData():

  Writes current data of Game Manager class which will be mentioned later to the created profile file in PlayerPrefs class.

4.1.7 Save Data Class

SaveData class is a simple structure to hold content of saving datas.

Fields

- int currentCoins:

  Holds number of coins players have.

- Dictionary<string,int> currentUpgradeData:

  This dictionary holds current upgrade names as string and their levels as int.

- List<DealerItemScript> currentItems :

  This container holds current weapons of the character.

Methods

- int GetCoins():

  Returns currentCoin value.

- Dictionary<string,int> GetCurrentUpgradeData():

  Returns currentUpgradeData dictionary.

- List<DealerItemScript> GetCurrentItems() :

  Returns currentItems list.

4.2 Level 1 Scene

This scene is the first level of the game where player starts his/her story and it is the environment where the main game loop exists. All game objects related with game mechanics and design is in this scene such as fighting mechanics, 2D physics controllers, input handlers, enemy mechanics and other items in game environment.

4.2.1 Level 1 Scene Classes

This Scene contains most of classes in the overall implementation of the project. It is hard to show every interaction in a single diagram so description of the classes will be handled by dividing the classes into main groups :

- Manager Classes :

  -GameManagerClass

  -SaveManager class

- Physics Handler Classes :

- RaycastController class

- Controller2D class

- Object Pooling Classes :

  -PoolManager class

  -PoolObject class

- Player Classes :
    - -PlayerInput class
    - -PlayerBehaviour class
    - -PlayerAttack class
    - -PlayerHurt class
    - -JetpackController class
    - -WeaponSwitcher class

- Enemy Classes :
    - -Enemy class
    - -PatrollingEnemy class
    - -ShooterEnemy class
    - -FlyerEnemy class
    - -KeyKeeperEnemy class
    - -EnemyHurt class

- Interactable Object Classes :
    - -Interactable class
    - -Checkpoint class
    - -VendingMachine class
    - -RewardChest class
    - -OrdinaryGate class
    - -BossGate class
- Trap Object Classes :
    - -Welder class

- Projectile Classes :
    - -Projectile class
    - -DefaultAmmo class
    - -HomingMissile class
    - -DropableBomb class
    - -BOStraightRocket class

- Weapon Classes :
    - -Weapon class
    - -BOStraightLauncher class
    - -BossMG class
    - -Turret class

- Reward Classes :
    - -Drop Master class

4.2.2 Manager Classes

Game Manager Class :

GM class is a singleton and also a component of a ‘DontDestroyOnLoad’ object. Main purpose of this class handling data transaction between scenes and holding current data to save of player.

Fields :

- int currentMultiverseCoins :

This field hold total value of coin collected from enemies by players.

- public Dictionary<string,int> currentUpgradesData;

Vending Machine objects enables players to upgrade their attriutes and buy new weapons In exchange of money. This dictionary holds current upgrade levels with name of attributes and level of the attributes.

- List<DealerItemScript> currentShopData :

This container holds current weapons in Vending Machine objects so that which item is purchased or equipped can be defined by this list.

- static GameManager instance :

Singleton instance of the class.

Methods :

- public static GameManager Instance :

  Get instance property.

- Public List<DealerItemScript> ShopData :

  Get currentShopData list.

- Void InitializeUpgradeData() :

  This method is called in Start() method of Unity. Predefined upgradable attributes are added to ‘currentUpgradesData’ field here.

- Void InitializeShopData() :

  This method also in called in Start() method. Predefined weapon which players can buy are added ‘currentShopData’ field here.

- Public int GetUpgradesDataItemLevel(string item) :

  Returns item level of given item name.

- Public int GetCoins() :

  Returns currentMultiverseCoins field.

- Public bool SpendCoins(int value) :

  When players try to buy new item, this method checks if they have got enough coins to buy that item. If they have got the coins, then value is subtracted from currentMultiverseCoins value and the method returns true. Otherwise, players do not have got enough coins so the desire item can not be bough and the method returns false.

Save Manager Class :

Save Manager class is mentioned in the previous section where it is used to load previous save datas. In Level 1 Scene when players quits from the game. Their current data where is held in Game Manager class gets written as a Save Data instance to a file. This files are seperated with different save profiles as mentioned in the first section.

4.2.3 Physics Handler Classes

These two classes are written to form smooth movement mechanics, handling collisions and minimize cost of built-in physics of Unity Engine.

Raycast Controller Class :

Main purpose of Raycast Controller class to create a custom collision for moving on, standing on and falling from platforms. Collision system in the class is based on bounds of BoxCollider2D object. Raycast Controller spawns raycast through each direction from bounds of the box collider so when these raycasts hits specified layer, it simulates the collision.

Fields :

- public LayerMask collisionMask : LayerMask class is a built-in class of Unity Engine. According to selected LayerMask in 'CollisionMatrix', the object with Raycast Controller class decides if it has permission to collide with that object. For example, when 'Terrain' is selected as collisionMask, the object is able to collide with that object but nothing else.

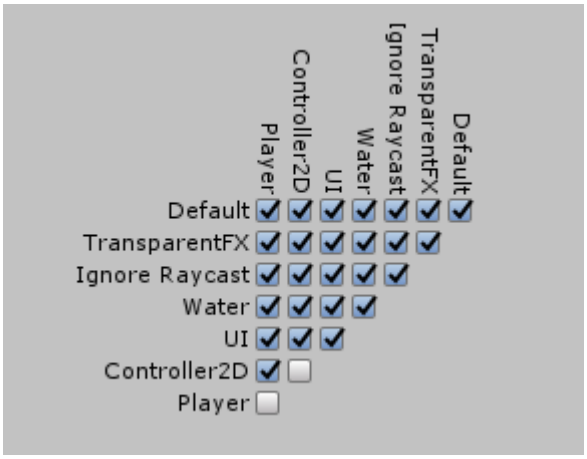


Fig. Collision Matrix in Unity

- public const float skinWidth = .015f :
- private const distanceBetweenRays = .25f :
- public int horizontalRayCount;
- public int verticalRayCount;
- public float horizontalRaySpacing;
- public float verticalRaySpacing;
- public BoxCollider2D col :
- public RaycastOrigins raycastOrigins;

Fields :

- public void CalculateRaySpacing() :

As mentioned previously, Raycast Controller spawns rays through four main directions. To achieve a proper collision checking, casting only one ray to the directions is not enough. Especially in tile-based games. In this method, according to desired distance between rays, how many rays can spawn from each side of the box collider is calculated. This calculation is done by simply dividing height and weight of the bounds by desired disstance between rays.

```
horizontalRayCount = Mathf.RoundToInt(boundsHeight / dstBetweenRays);
verticalRayCount = Mathf.RoundToInt(boundsWidth / dstBetweenRays);

horizontalRaySpacing = bounds.size.y / (horizontalRayCount - 1);
verticalRaySpacing = bounds.size.x / (verticalRayCount - 1);
```

Fig. Calculation to find total ray count to spawn

- public void UpdateRaycastOrigins() :

This method finds corners of the box collider object, then sets attributes of 'raycastOrigins' field which is used to decide where to begin to spawn the rays.

```
raycastOrigins.bottomLeft = new Vector2(bounds.min.x, bounds.min.y);
raycastOrigins.bottomRight = new Vector2(bounds.max.x, bounds.min.y);
raycastOrigins.topLeft = new Vector2(bounds.min.x, bounds.max.y);
raycastOrigins.topRight = new Vector2(bounds.max.x, bounds.max.y);
```

Fig. Setting raycast origins with corners of the box collider.

Structs :

- public struct RaycastOrigins :

This is RaycastOrigins type mentioned before. This struct is used only in RaycastController class.

-public Vector2 topLeft, topRight :

Positions of top left and top right corners of the box collider object.

-public Vector2 bottomLeft, bottomRight :

Positions of bottom left and bottom right corners of the box collider object.

Controller 2D Class :

Controller2D is child class of RaycastController. Custom collision detection with raycast and movement mechanics are handled in this class.

Fields :

- public float fallingThroughPlatformResetTimer = 0.1f :

Some platforms can be passed through by players and players can also fall through in purpose. This is the period when standart collision detection starts working after falling inside a platform.

- Private RigidBody2D body :

A built in class of Unity Engine for making objects affected by 2D physics simulation.

- Public CollisionInfo collisions :

This struct holds current status of collision of raycasts. For example, if the rays which casted to the right direction hits something in their layermask then 'right' field of this struct becomes 'true'.

- Public Vector2 playerInput :

This is the two variable input vector which comes directly from PlayerInput class to move characterss.

Methods :

- private void HorizontalCollisions(ref Vector2 moveAmount) :

This method checks horizontal collisions which are the rayscasts to the right and the left directions. The method first decides facing direction of the player, then determines length of the rays according to current move speed so that if character goes so fast, it will not cause any collapsing situation between objects and characters. After that, according to desired raycount which is calculated in RaycastController class, starts to spawn raycasts to the facing direction. If the facing direction value is -1 then raycasts spawn to the left. Otherwise the value is 1 and it means raycasts will spawn to the right. If raycasts hit something then the collision info will be updated.

```
float directionX = collisions.faceDir;
float rayLength = Mathf.Abs(moveAmount.x) + skinWidth;

Vector2 rayOrigin = (directionX == -1) ? raycastOrigins.bottomLeft : raycastOrigins.bottomRight;
rayOrigin += Vector2.up * (horizontalRaySpacing * i);
RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.right * directionX, rayLength, collisionMask);
if (hit)
{
collisions.left = directionX == -1;
collisions.right = directionX == 1;

}
```

- Private void VerticalCollisions(ref Vector2 moveAmount) :

This method is the vertical version of HorizontalCollisions method.The method first decides current direction of character in Y axis, then determines length of rays according to current speed on Y axis so that if character falls or flies so fast, it will not cause any collapsing situation. After that, according to desired raycount which is calculated in RaycastController class, starts to spawn raycasts to the current Y direction. If the direction values is -1 then raycasts spawn to the bottom, Otherwise the value is 1 and it means raycasts will spawn to the up. If raycasts hit something then the collision info will be updated.

```
float directionY = Mathf.Sign(moveAmount.y);
float rayLength = Mathf.Abs(moveAmount.y) + skinWidth;

Vector2 rayOrigin = (directionY == -1) ? raycastOrigins.bottomLeft : raycastOrigins.topLeft;
rayOrigin += Vector2.right * (verticalRaySpacing * i + moveAmount.x);
RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.up * directionY, rayLength, collisionMask);
if (hit)
{
collisions.below = directionY == -1;
collisions.above = directionY == 1;

}
```

- Public void Move (Vector2 moveAmount, Vector2 input) :

When the player sends an axis input to move the character and when enemy characters wandering around, this method is called in Update() method. When method is called firstly, raycast origins from RaycastController updated. Then X component of the input vector is checked, if it is not zero then facing direction gets set. After that HorizontalCollisions method is called with value of ‘moveAmount.X’ input and horizontal collisions get updated. Then Y component of the input vector is checked, if it is not zero, then VerticalCollisions method is caleld with value of ‘moveAmount.Y’ input and vertical collisions get updated. Afther all collisions are set, MovePosition method of built in RigidBody2D class is called with moveAmount value and character moves according to that value.

▪

```
UpdateRaycastOrigins();
collisions.Reset();
if (moveAmount.x != 0)
{
    collisions.faceDir = (int)Mathf.Sign(moveAmount.x);
}
HorizontalCollisions(ref moveAmount);
if (moveAmount.y != 0)
{
    VerticalCollisions(ref moveAmount);
}
body.MovePosition(body.position + moveAmount);
```

4.2.4 Player Classes

Player classes are responsible for getting inputs, using physics handler classes, attacking of main character, hurting, using flying mechanics and weapon handling.

Player Input Class :

PlayerInput is a singleton class is gets actual input from keyboards/gamepads through RewiredInputManager class which is an external asset for to support all kind of controllers. Processing input, interaction between different UI's and different behaviors of buttons are implemented here.

Fields :

- public bool inUIinteraction :

  If player are using in-game user interface, buttons will work differently. For example, 'attack' button can be used as 'purchased' button in shop interface. This boolean flag holds the related info.

- Public bool inObjInteraction :

  Purpose of this flag is same with the previous one but this is not for user interfaces. It hold the info about if the player in interaction in a interactable object such as a gate or reward chest.

- Public GameObject interactionObject :

  What UI object or in-game object player is in interaction with.

- Private PlayerBehaviour player :

  PlayerBehaviour class reference for passing movement input to physics handler classes which are mentioned before.

- Private AnimationHandler animHandler :

  AnimationHandler class reference for handling running animation of characters.

- Private bool fireInput :

  Is 'fire' tagged button in Rewired Input Manager is down?

- Private bool jetpackInput :

  Is 'jetpack' tagged button in Rewired Input Manager is down ?

- Private bool jetpackJustTurnedOff :

  Is 'jetpack' tagged' button in Rewired Input Manager is up ?

- Private bool backButton :

  Is 'back' tagged button in Rewired Input Manager is down? This is used for user interfaces.

- Private bool startButton :

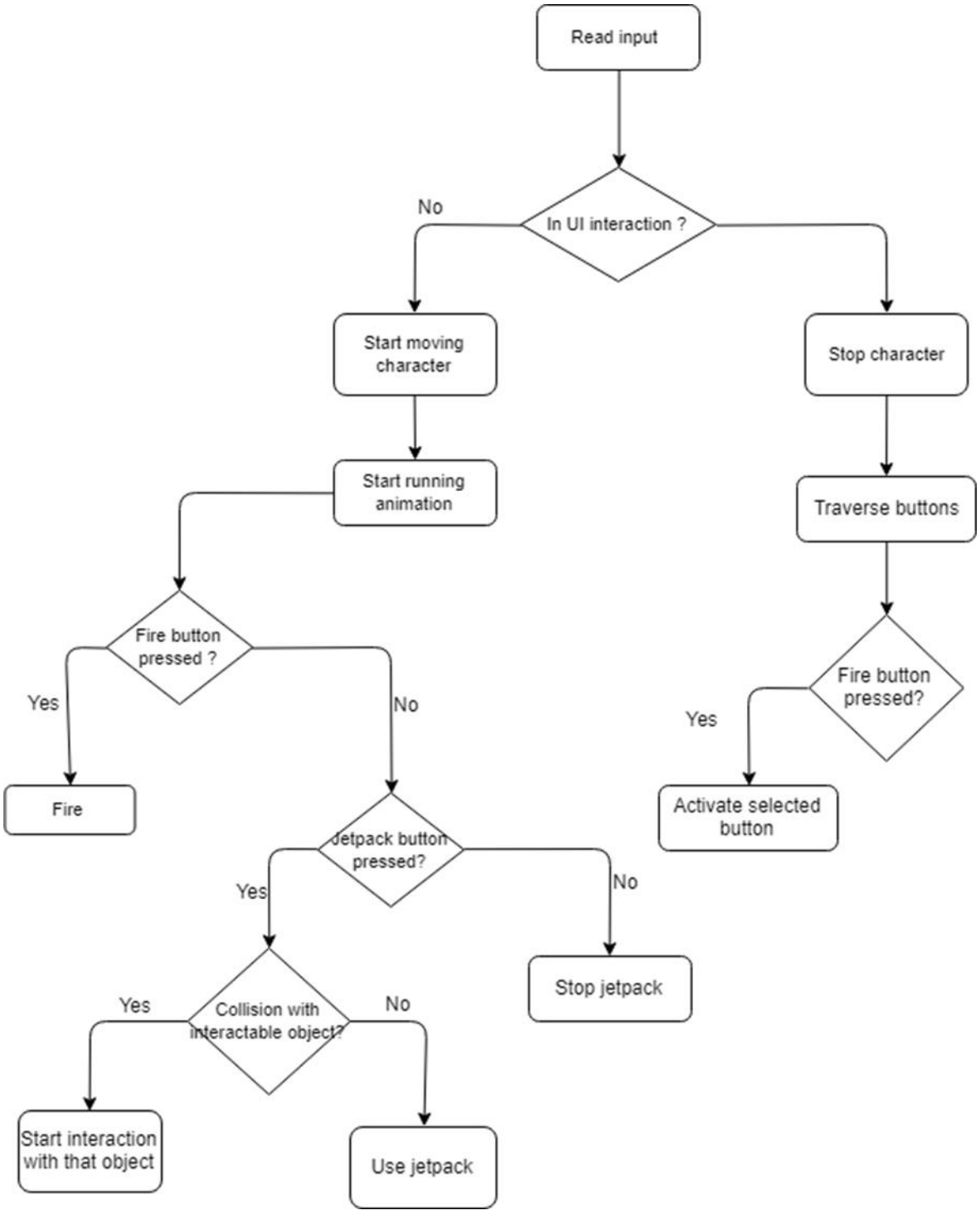
  Is 'start' tagged button in Rewired Input Manager is down?

Methods :

- GetInput() :

  This is the prior method in PlayerInput class. It reads current input from Rewired Input Manager then sets the input into boolean fields then process it in statements. Then the method checks if there is an open

interface. If there is no open user interface, then input movement vector is set and running animation started. In this statement if player pushes ‘fire’ tagged button in the input controller, then character fires. If the character is colliding with an interactable object ‘fire’ tagged button interacts with it. Let’s get back to first ‘if’ statement. If there is an open user interface, then axis input is used to traverse buttons and ‘fire’ tagged button to select current button.



Player Behaviour Class :

Player Behavior is the class that reflects all input reaction, physics and collision detetion on the character in actual scene. All mechanics which mentioned before called in FixedUpdate method of this class.

Fields :

- private float moveSpeed :

  Speed multipler for horizontal axis of velocity vector.

- Private float gravity :

  Gravity force implies to the character when he falls.

- Private Vector3 velocity :

Final movement vector of the character. Velocity is simply multiplication of directional input with few other factors.

- Private Controller2D controller :

Controller2D reference for Move method.

- Private Vector2 directionalInput :

  Raw input vector comes from Player Input class.

- Private bool facingRight = false :

  Which direction the character facing.

- Private PlayerAttack attackBehaviour :

  PlayerAttack class reference.

- Private JetpackController jetpackBehavior :

  JetpackController class reference.

Methods :

- public void SetDirectionalInput (Vector2 input) :

  This class is a setter for ‘directionalInput’ field. PlayerInput class pass its reference to here by using this method. According to X component of the input, character sprite is mirrored. If X value is positive then the character sprite faces right, otherwise X value is negative then the character sprite spins to left.

- Public void OnJetpackInputDown() :

  This method is also called from PlayerInput class. When method is called StartJetpack of JetpackController class which is explained soon is invoked to start jetpack animations and Y component velocity is calcualted and set according to jetpack speed multiplier level.

- Public void OnJetpackInputDown () :

  Stops the jetpack by invoking StopJetpack method of JetpackController class which stops animations and jetpack boosting.

- Public void OnFireInputDown () :

  Fires current weapon by calling Fire method of PlayerAttack class.

- Private void CalculateVelocity() :

  This is the method determines the final velocity of the character before calling Move method of Controller2D class which is the method actually moves the player on screen.

  Firstly, X component of the vector is calculated by multiplying X component of directional input with moveSpeed and AfterburnerSpeedMultiplier value of Jetpack Controller. (AfterburnerSpeedMultiplier is horizontal power of the jetpack and it is upgradable. When it is upgraded main character’s move speed increases.) Secondly, Y component of the vector is increased by gravity value to apply gravitial force to the character.

```
float targetVelocityX = directionalInput.x * moveSpeed * jetpackBehaviour.AfterburnerSpeedMultipler;
velocity.x = Mathf.SmoothDamp(velocity.x, targetVelocityX, 1f);
velocity.y += gravity * Time.deltaTime;
```

Player Attack Class :

When players send input to shoot with their weapons PlayerInput class calls Fire method of this class. This class is a holder for weapon behaviour.

Fields :

- public Weapon weaponBehavior :

  Weapon class reference.

Methods :

- public void Fire() :

  Fire method first checks if current gun has enough ammo to shoot from singleton class WeaponSwitcher. If the ammo is enough then ammo count decreases, fire sound plays and current weapon fires.

```
if(WeaponSwitcherScript.Instance.UseAmmo(weaponBehaviour.weaponName))
{
    SoundHandlerScript.Instance.PlaySound(2);
    weaponBehaviour.Fire();
}
```

- public void ChangeEquippedWeapon (Weapon newWeapon) :

This method is kind of setter for weaponBehaviour field. WeaponSwitcher class which will be mentioned later uses this method to switch back different method according to player input.

Player Attack Class :

This class handles getting hurt and getting healed mechanics for the main character and its a child of HittableObject class.

HittableObject class is interface-like class to use GetHurt method generically among main character and enemies. HittableObject class has two protected float fields which are ‘currentHealth’ and ‘maxHealth’. ‘currentHealth’ field is the dynamic value of object’s hitpoint. In the other hand ‘maxHealth’ is max limit of hitpoint an object hold.

HittableObject class has only one virtual method GetHurt(). The method has two overload which are GetHurt(Weapon weapon) and GetHurt(float amount).

Fields :

- private float ‘currentHealth’ :

  Inherited current health field from HittableObject class.

- Private float ‘maxHealth’ :

  Inheritd max health field from HittableObject class.

Methods :

- public override void GetHurt (Weapon weapon) :

GetHurt methods in all overloads have same goal : to decrease health and check ‘currentHealt’ is below zero or equal. If ‘currentHealt’ equals to zero then kill the main character then spawn him if there is a available checkpoint. Otherwise, restart level.

In this overload, Weapon reference given as an input then ‘damage’ field of weapon is taken and substracted from ‘currentHealth’ and pop-up text notifies players about the damage. After that, ‘currentHealth’ is check if it is below zero or equal. If it is then an active spawn point is requested from IngameLevelController class. If spawn point is not (0,0) then main character spawned there. Otherwise, whole level gets restarted again.

- Public override void GetHurt (float amount) :

This method has almost the same implementation with the previous one. The only difference is without giving an mediator Weapon class, getting hurt is handled with only raw amount value.

- Public void RefreshLife() :

Fills the ‘currentHealth’ to max limit.

- Public void GetHeal (int healAmount) :

According to given heal amount the method checks if ‘currentHealth’ plus ‘healAmount’ less than or equals to ‘maxHealth’, ‘currentHealth’ value increased by ‘healAmount’. Otherwise, maximum value of ‘currentHealth’ can be ‘maxHealth’. After these statements, this info passed to TowinHpDisplayer class which is responsible for to display health points on UI.

Jetpack Controller Class :

This class controls particle effects of a jetpack and external speed multiplier variables to sustain upgradable speed mechanic for main character.

Fields :

- private ParticleSystem jetpackParticles :

A built-in particle system of Unity Engine for particle effects.

- Private int jetpackFlyingLevel = 1 :

Vertical speed booster for player. If this level gets upgraded then main character flies faster on Y axis.

- Private int jetpackAfterburnerLevel = 1 :

Horizontal speed booster for player. If this level gets upgraded then main character flies and runs faster on X axis.

Methods :

- public void UpdateData() :

Gets current upgrade data from GameManager script. If there are changed in levels fields should be updated before start method.

```
jetpackFlyingLevel = GameManagerScript.Instance.GetEngineerDataItemLevel ("flyengine");
jetpackAfterburnerLevel = GameManagerScript.Instance.GetEngineerDataItemLevel ("afterburner");
```

- Public void StartJetpack() :

First, UpdateData method is called, then if ‘jetpackParticles’ does not emitting, Play method of ‘jetpackParticles’ called.

- Public void StopJetpack() :

If ‘jetpackParticles’ system is emitting (animation playing), then call Stop method of ‘jetpackParticles’.

- Public float FlyingSpeedMultipler () :

Returns a calculation which used to determine a speed multiplier to increase Y component of character speed vector by using ‘jetpackFlyingLevel’ value.

```
get{return 3.5f / (1+(jetpackFlyingLevel * 0.15f));}
```

- public float AfterBurnerSpeedMultiplier () :

Returns a calculation which used to determine a speed multiplier to increase X component of character speed vector by using ‘jetpackAfterburnerLevel’ value.

```
get{return (1 +(jetpackAfterburnerLevel * 0.15f));}
```

Weapon Switcher Class :

Weapon Switcher singleton class holds current weapons of and ammo types for them for main character. Players can switch between their weapons through this class.

Fields :

- private Dictionary<string,Weapon> weaponSlots :

A container for current Weapon objects in the scene. Weapon names are key and Weapon class reference is value in a pair.

- Private Dictionary<string,int> weaponAmmoHollder :

A container for ammo count for each specific weapon. Weapon names are key and ammo count is value in a pair.

- Private List<string> weaponNames :

Helper weapon names list to traverse the dictionaries easier.

- Private int currentWeaponIndex = -1 :

This variable holds ‘weaponNames’ list index of current equipped weapon. Value -1 means no weapon in hands of main character.

Methods :

- private void InitializeAmmoCount(string weaponName) :

This method is used when new weapon is being added to weaponSlots. According to weapon name different base values are set to dictionary. For Assault Rifle ammo value is -1 which means infinite and Rocket Launcher 5.

```
switch(weaponName)
{
    case "assaultrifle":
        weaponAmmoHolder[weaponName] = -1;
        break;
    case "rocketlauncher":
        weaponAmmoHolder[weaponName] = 5;
        break;
    default:
        Debug.LogError("Weird weapon name.");
        break;

}
```

- public void AddWeapon (string weaponName) :

Purpose is adding new weapon into ‘weaponSlots’ dictionary. Firts, the method checks if ‘weaponSlots’ already contains input string. If it does not contain then built-in method Instantiate is called to create the desired weapon from Resources.Load(weaponName) method of Unity Engine. Then the instantiated Weapon object added into ‘weaponSlots’ dictionary and InitializeAmmoCount method is called to determine base ammo count. After that, the Weapon object in Unity hierarchy becomes child of ‘player’ object so it can move with ‘player’ object’s sprite. Finally, if this weapon is the first to be added then ‘currentWeaponIndex’ becomes 0.

```
if(!weaponSlots.ContainsKey(weaponName))
{

    var newBornWeapon = Instantiate (Resources.Load (weaponName), transform.position, Quaternion.identity) as GameObject;
    weaponSlots.Add (weaponName,newBornWeapon);
    weaponNames.Add (weaponName);
    InitializeAmmoCount(weaponName);
    newBornWeapon.transform.SetParent (transform);
    if(currentWeaponIndex == -1)
    {
        currentWeaponIndex = 0;
    }
    else
    {
        newBornWeapon.SetActive(false);
    }
}
else{
    Debug.LogError ("Weapon : " + weaponName + " is already equipped.");
    return;
}
```

- public void SwitchForward () :

This method enables players to switch to next weapon in ‘weaponSlots’ dictionary. When method is called current Weapon object gets deactivated from the game scene. Then ‘currentWeaponIndex’ value is increased by one. Increment is done in cycle by weapon size. For example, there is three different weapons in the list. The player switches to the last weapon. If the player tries to switch forward again then index will go to the first item again. After indexes handled then RefreshSwitcher method is called.

- Public void RefreshSwitcher ():

When RefreshSwitcher method is called Weapon object in the game scene with current index gets activated and PlayerAttack class is notified about current Weapon reference change by calling ChangeEquippedWeapon (Weapon) method of PlayerAttack class. After this settings UIWeaponImageDisplayer class is notified about changes and displays current weapon on player in-game interface.

```
var temper = weaponSlots [weaponNames [currentWeaponIndex]] as GameObject;
temper.SetActive(true);
playerAttackBehavior.ChangeEquippedWeapon (temper);
UIWeaponImageDisplayerScript.Instance.SetWeaponSprite(temper.GetComponent<SpriteRenderer>().sprite);
UIWeaponImageDisplayerScript.Instance.UpdateAmmoText(weaponAmmoHolder[weaponNames[currentWeaponIndex]] == -1 ? "INF"
    : weaponAmmoHolder[weaponNames[currentWeaponIndex]] + "");
```

- public void UseAmmo (string weaponName) :

As mentioned before this method called in PlayerAttack class. When players send input to fire weapon this method is called. The method checks if there is enough ammo for a shoot. If there is enough ammo then decreases value of given weapon name in ‘weaponAmmoHolder’ dictionary. Then calls RefreshSwitcher to update the user interface and returns with ‘true’. If there is no ammo to fire, then the method simply returns false.

```
if(weaponAmmoHolder[weaponName] - 1 >= 0)
{
    weaponAmmoHolder[weaponName] -= 1;
    RefreshSwitcher();
    return true;
}
else
{
    return false;
}
```

4.2.5 EnemyClasses

Enemy Classes contains enemy character’s specific and generic behaviours such as moving attacking and dying.

Enemy Class :

This class has same functionality as PlayerBehaviour. However, this class is a child class of PoolObject class because enemies are pool objects and they should be reused in some parts of the game.

Fields :

- public float patrolSpeed = 0.25f :

  Most of the enemies are patrolling between the start and the end point of platforms. This variable is the speed multiplier for their movement.
- For all another field see PlayerBehaviour class.

Methods :

- See PlayerBehaviour class.

PatrollingEnemy Class :

PatrollingEnemy class is a child class of Enemy class. Game objects with this component on them patrols platforms between the start point and the end point of a platform. To achieve behaviour raycasts are used.

Fields :

- public float checkDistance :

  Height of raycast to decide if enemy is at the end of a platform.

Methods :

- protected void CheckEdgeFall (Vector2 moveAmount) :

This method simply spawns a raycast in front (0.5f) of enemy object which is headed downwards. As long as the raycasts hits something (which means there is ground in front of the object), the enemy object keep its current velocity. Otherwise, if the raycast hits nothing (which means there is an edge in front of the object), then current movement horizontal direction of current velocity vector is set to opposite value by multiplying with -1.

```
Vector2 edgeRayOrigin;
if (facingRight)
{
    edgeRayOrigin = new Vector2(controller.raycastOrigins.bottomRight.x + 0.5f, controller.raycastOrigins.bottomRight.y) + (Vector2.right * moveAmount.x);
}
else
{
    edgeRayOrigin = new Vector2(controller.raycastOrigins.bottomLeft.x - 0.5f, controller.raycastOrigins.bottomLeft.y) + (Vector2.right * moveAmount.x);
}
Debug.DrawRay(edgeRayOrigin, Vector2.down * checkDistance, Color.blue);
RaycastHit2D hitEdge = Physics2D.Raycast(edgeRayOrigin, Vector2.down * 5, checkDistance, controller.collisionMask);
if (!hitEdge)
{
    patrolSpeed *= -1;
    SetDirectionalInput(new Vector2(patrolSpeed, 0));

}
```

Code fragment : CheckEdgeFall method

- protected virtual void Patrol() :

Encapsulation method for CheckEdgeFall and Move method of Controller2D class. This method is called in overriden Update method.

```
controller.Move(velocity * Time.deltaTime, directionalInput);
CheckEdgeFall(velocity * Time.deltaTime);
```

Code fragment: Patrol method

```
protected override void Update()
{
    base.Update();
    Patrol();
}
```

Code fragment: Override Update method of PatrollingEnemy class

ShooterEnemy Class :

This class is a child class of PatrollingEnemy class. In this class behaviour patrolling objects also fires weapons which they equipped.

Fields :

- public GameObject weaponToEquip :

  Weapon prefab which the enemy is going to use.

- Protected Weapon weaponBehaviour :

  Weapon class reference of instatiated prefab.

- Protected float attackCooldown :

  Every ‘attackCooldown’ seconds a shooter enemy fires its gun.

- Protected bool ableTofire = true :

  When the enemy fires its gun this flag becomes false until ‘attackCooldown’ seconds passed.

- Protected float timeAfterFiring :

  Seconds passed after the enemy’s attack. This value is going to be compared with ‘attackCooldown’ to determine if the enemy is able to fire.

Methods :

- public override void OnObjectReuse () :

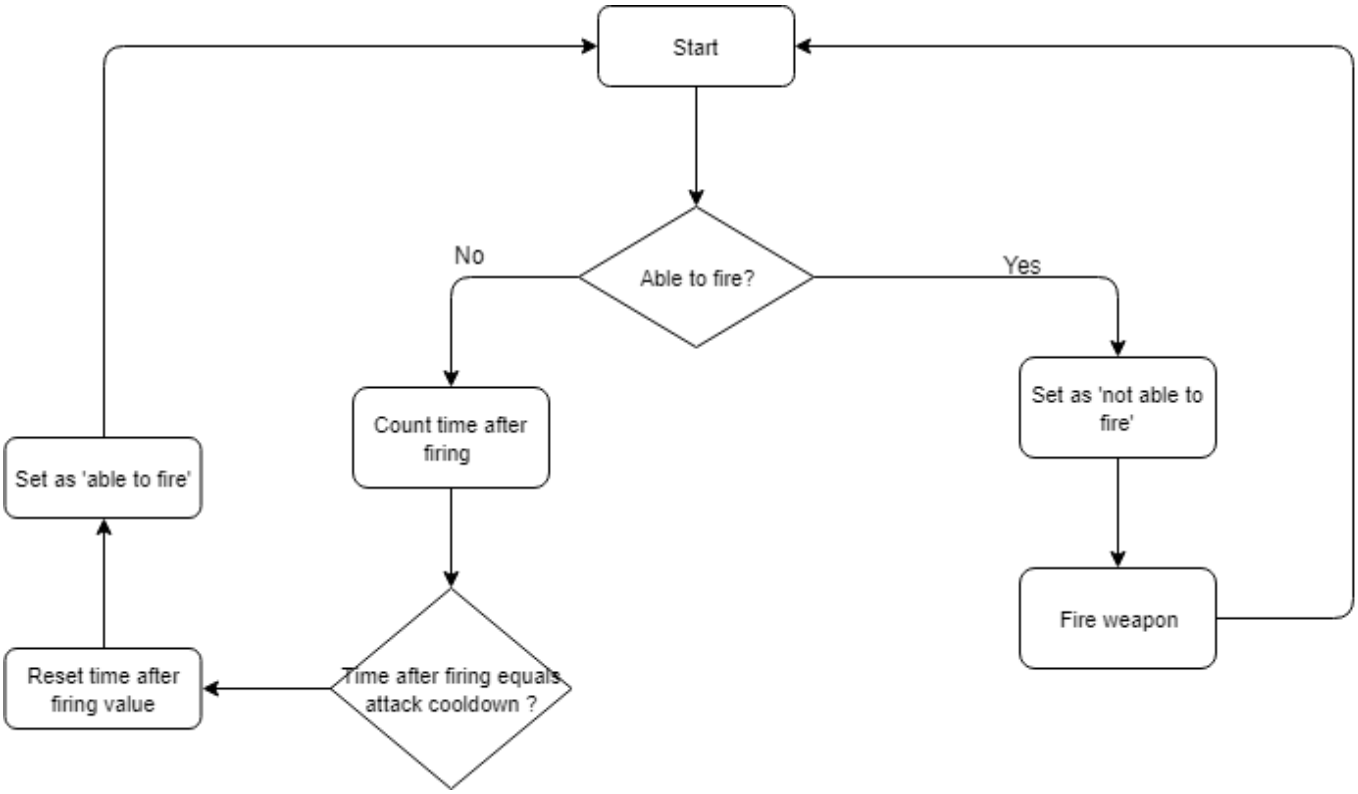
  This method is the configure method of PoolObject class. It is called when object reused from the object pool. It is also called in Start method of manual created objects.

  In the method given weapon object prefab is instantaited and its Weapon class reference assigned to ‘weaponBehaviour’ field. Then the instantiated weapon is set as child of the enemy object.

```
equippedWeapon = Instantiate (weaponToEquip, transform.position, Quaternion.identity);
weaponBehaviour = (WeaponScript)equippedWeapon.GetComponent(typeof(WeaponScript));
equippedWeapon.transform.SetParent (transform);
```

Code fragment: OnObjectReuse method of ShooterEnemy class

- protected void FireControl () :



Control mechanism for attack cooldown. If ‘ableToFire’ value is true, then Fire method of ‘weaponBehaviour’ reference is called. Then ‘ableToFire’ value gets set to false. If ‘ableToFire’ value is not ‘true’, then it means the enemy object fired and it is waiting for cooldown so ‘timeAfterFiring’ value is increased by ‘Time.deltaTime’ value of Unity Time Manager class. If ‘timeAfterFiring’ value is greater or equal than ‘attackCoolDown’ value, then ‘ableTofire’ vale is set to true and ‘timeAfterFiring’ value gets reset.

FlyerEnemy Class :

This class is forms a behavior for flying enemy objects ‘flyers’. This class is not a child class of Enemy class but Weapon class. Because this class does not require moving on platforms and custom collision and behaves like a weapon itself.

Fields :

- private Vector2 direction= Vector2.right :

  When a scene starts all flyers fly towards the right. According to their move speed phase difference is determined.
- Public float moveSpeed :

  Move speed multipler of FlyerEnemy object.
- Private Rigidbody2D flyerBody :

  Rigidbody2D component of the object.
- Pivate Transform deliveryTarget :

Transform component of target which is going to be bombed.
- Private bool permissionToDrop :

  Flag to check if statements are set to drop a bomb.
- Privat e deviation= 2 :

  Deviation with exact target position as units. X coordinate deviates by given value.
- Private float timePassed = 0 :

Time passed after flyer starts its one-way flight.

Methods :

- protected override void Start () :

  In Start method ‘deliveryTarget’ field is set to ‘player’ object on the scene by getting singleton instance’ position of PlayerInput class. Then Start method of parent class is called.

```
flyerBody = GetComponent<Rigidbody2D>();
deliveryTarget = PlayerInput.Instance.gameObject.transform;
base.Start();
```

Code fragment: Start method for FlyerEnemy class

- private float CalculateReturnTime () :

  Returns one way flying duration to the end horizontally. Based on calculation of ‘if a flyer flies to the end of screen with moveSpeed of 6 in 6 seconds then proportion is ‘36 / moveSpeed’.

```
return 36 / moveSpeed;
```

- Private void CheckDeliveryPoint () :

The enemy object checks if it is exactly top of the target according to deviation. If the enemy object's current position plus or minus deviation is close to target's position and 'permissionToDrop' flag is set to 'true', then DropBomb method is called.

```
if(!permissionToDrop)
{
    return;
}
if(transform.position.x + divergence > deliveryTarget.position.x && transform.position.x - divergence < deliveryTarget.position.x)
{
    DropBomb();
}
```

Code fragment: CheckDeliveryPoint method of FlyerEnemy class

- private void DropBomb() :

This method sets 'permissionToDrop' value to false and calls generic Fire method of base Weapon class.

- Private void FixedUpdate () :

In FixedUpdate method, MovePosition method used to move the enemy object according to its direction and move speed field. In addition to that, 'timePassed' value increased over time. If 'timePassed' value greater than the value which is returned by CalculatedReturnTime, then the flyer object changes its direction, 'permissionToDrop' value gets set to 'true' and 'timePassed' value is reset.

KeyKeeper Class :

KeyKeeper class is a simple sprite mirror configuration for 'keykeeper' object in the scene. The object turns its face according to position of its target.

Fields :

- private Transform targetTransform :

Transform component of a potential target.

Methods :

- private void Start() :

In Start method 'tergetTransform' field is set to 'player' object on the scene by getting singleton instance' position of PlayerInput class.

- private void Update() :

In Update method loop X coordinates of 'keykeeper' object and targets are compared. If target stays right side of 'keykeeper' object, then 'localScale' of the key keeper sets to Vector2.one (changing

X component of ‘localScale’ makes object mirroed in Unity). If target stays left side of the key kepeper, then X component of ‘localScale’ is set to -1.

EnemyHurt Class :

EnemyHurt class has almost same functionality as PlayerHurt class and it is also a child of HittableObject class. It controls hurting and dying behaviours of enemy objects.

Fields :

- public ParticleSystem bloodSplash :

  Dying effect particles as built-in ParticleSystem component of Unity Engine.
- Private SpriteRenderer spriteRenderer :

  Built-in SpriteRenderer component of object.
- Public float hurtDuration = 0.3f :

  Paints enemy object color of red in ‘hurtDuration’ seconds.

Methods :

- public override void GetHurt (WeaponScript weapon) :

  When enemies are hit, according to ‘damage’ field of ‘weapon’ object their ‘currentHealth’ are decreased and HitEffect coroutine gets started. If their ‘currentHealth’ equals or less than zero, then DeathProcedure method is called.
- IEnumerator HitEffect (float hurtEffectDuration) :

  After enemy object is hit, its sprite colord gets red by assigning its ‘color’ field of SpriteRenderer class as red. Then, coroutine waits for ‘hurtEffectDuration’ seconds and turns its color back to default.
- Private void DeathProcedure () :

  This method contains set of operations before deactivating dead enemy object. First, ‘bloodSplash’ particles instantiated at the position of the enemy object. Then as rewards coins and random items are requested from DropMaster singleton class. After playing killing sound for the enemy object, the object gets deactivated for future use by calling UnUse method of parent Enemy class object.

```
IngameLevelControllerScript.Instance.DropItem(transform.position);
Instantiate(bloodSplash, transform.position, Quaternion.identity);
SoundHandlerScript.Instance.PlaySound(1);
CoinDropperScript.Instance.CasualDrop(transform.position);
(gameObject.GetComponent(typeof(Enemy)) as Enemy).Unuse();
```

4.2.6 Interactable Objects Classes :

Interactable Class :

A virtual class for overall interactable objects in the game.

Methods :

- public virtual void Interact () :

  Any operation when interacted.

- Public virtual void UnInteract () :

Stop interaction with the object.

Checkpoint Class :

Checkpoint class is implements behaviour of ‘checkpoint’ objects in the game scene.

Fields :

- private bool isActivated = false :

  Did the player activate the checkpoint.

- Public int checkpointIndex = -1 :

  Unique index for ‘check point’ object. As default checkpoints have index of -1. InGameLevelController class assigns the indexes when level is loaded and last index also notified to GameManager class.

- Private Animator animator :

  Built-in Animator class component for controlling flag animation.

Methods :

- public override void Interact () :

  When players send interaction request ‘isActivated’ value gets set as ‘true’ and ‘checkPointIndex’ value is passed to InGameLevelController class. Finally, “activation” animation of ‘checkpoint’ object is triggered to start.

OrdinaryInGameGate Class :

This class implements functionality of gates in the scene. Players needs to find the key to Boss Level from ‘key keeper’ and they need to find the correct gate. Only one gate contains the key keeper and other content is created randomly according to gate number in the scene. It is a child class of Interactable class to use Interact method when players send request for interaction.

Fields :

- public int contentType = 0 :

  This field determines what is inside the gate. default is no content, 1 key keeper enemy, 2 reward, 3 spawn enemy wave. This value is set on loading the scene by IngameLevelController class.

- Public float gateOpeningTime = 2 :

  Openning animation and activating duration of a gate object.

- Private bool gateUsed = false;

  If gate is used then this flag becomes ‘true’ so that main character can not interact again.

- EnemySpawner enemySpawner :

EnemySpawner class reference which is used to spawn enemies from PoolManager at current position.

- Animator animator :

  Animation controller component for ‘ordinarygate’ object.

Methods :

- private void SpawnReward () :

  Spawns a RewardChest object at current position.

- Private void SpawnKeyKeeper () :

  Calls SendOne (int) method of its EnemySpawner class with value of 1. Index of 1 represents a ‘keykeeper’ prefab for the spawner.

- Private void SpawnEnemyWave () :

  Calls SendPack (int,int) method of its EnemySpawner class with value 0 for ‘default’ enemy prefab and 5 for enemy spawn count.

- Private IEnumerator ActivateContent () :

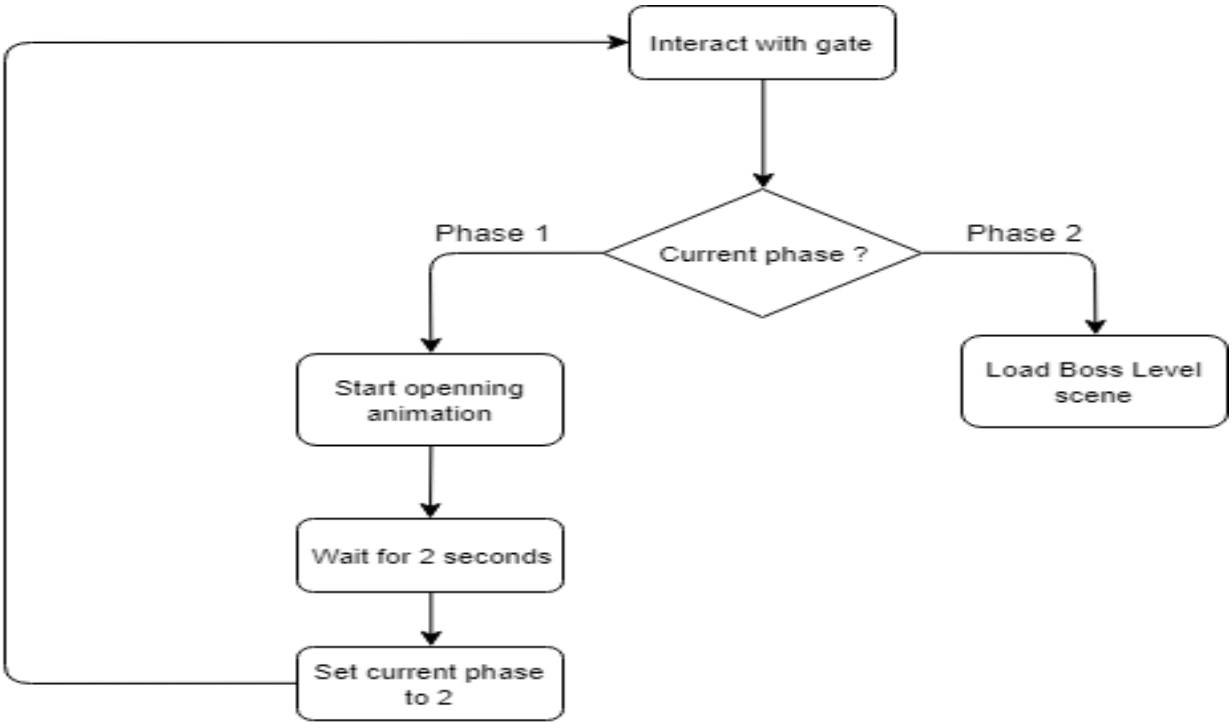
  Enurmerator for opennig gate and revealing gate content. In gate openning process, ‘gateUsed’ value gets ‘true’ firstly. Then, BoxCollider2D component gets disabled to prevent future unnecessary collisions. After that, ‘opengate’ animation tag is triggered and coroutine waits for ‘gateOpeningTime’ seconds. Finally, a switch structure decides which content is going to be activated according to ‘contentType’ value.

- Public override void Interact () :

  Interact method which is inherited from Interactable class. In this method ‘gateUsed’ flag is checked. If the gate is not used then ActivateContent coroutine is started.

BossGateClass Class :

Implementation class for behavior of the gate which takes players to Boss Level. If players killed the key keeper enemy, then they got the key to this gate.



Fields :

- private int currentPhase = 1 :

  Current state of the gate object. If current phase is 1, then the gate is closed yet. If current phase is 2, then the gate is open and another interact leads player to go to Boss Level.

- Animator animator :

  Animation controller for ‘bossgate’ object.

Methods :

- private IEnumarator OpenGate () :

  Coroutine for openning the gate which is similar to ActivateContent coroutine of OrdinaryIngameGate class. Firstly, ‘open’ animation is triggered. Then coroutine waits for 2 seconds and sets ‘currentPhase’ value as 2.

- private void GoToBossLevel () :

  LoadScene method of built-in SceneManager class is called with scene name ‘boss’.

- Public override void Interact () :

  Interaction method for players. According to switch mechanism in the method, if ‘currentPhase’ value is 1, then OpenGate coroutine is called. Otherwise which means ‘currentPhase’ value is 2, then GoToBossLevel method is called.

4.2.7 Trap Objects Classes :

Welder Class :

Welder is an environmental trap object which spawns a deadly beam. It is automatically switches off and on itself. When it is working it creates a line between the first obstacle which represents the beam.

Fields :

- public int targetAxis :

  Which axis the beam is casted. Value 1 for right, -1 for left, 2 for up and -2 for down directions.

- Private Vector2 targetVector :

  Direction vector which is determined according to ‘targetAxis’ value.

- Private LineRenderer lineRenderer :

  A built-in component of Unity Engine which draws lines in the scene.

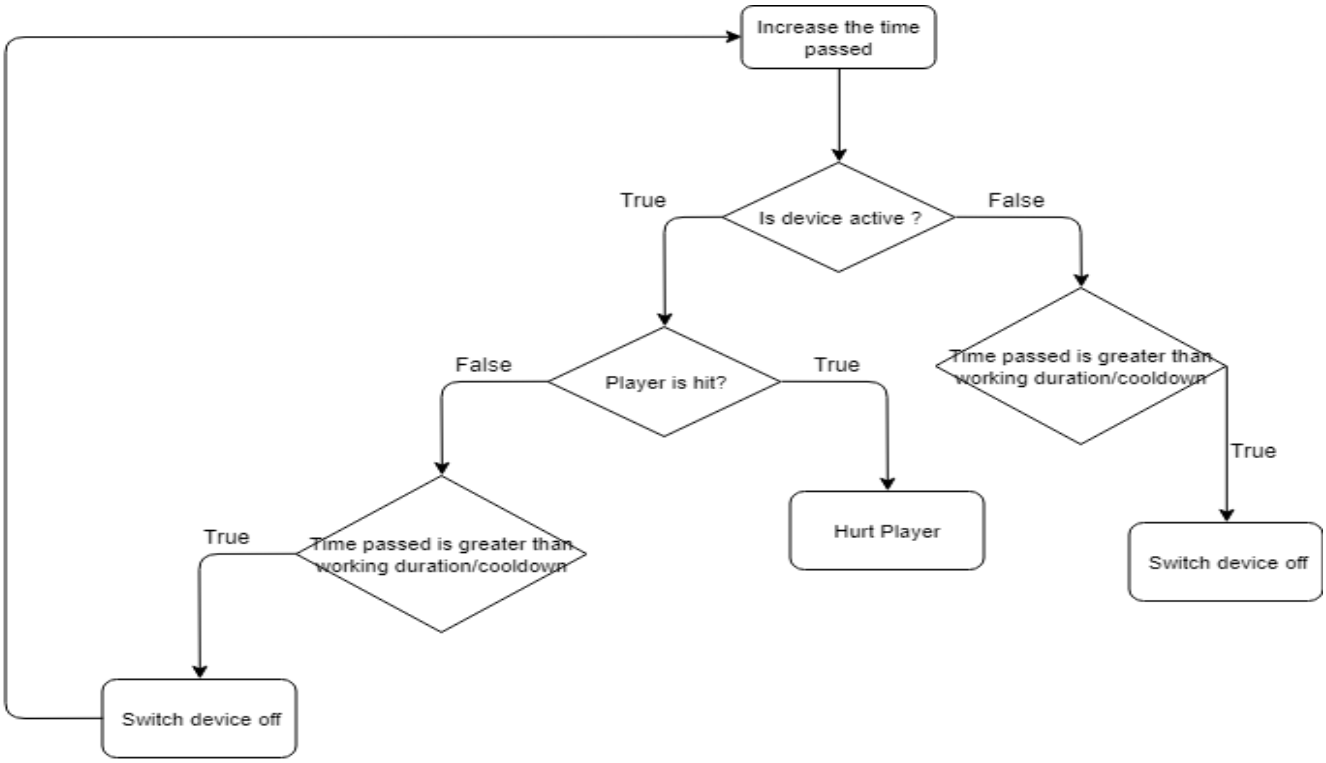
- Private bool deviceActive = false :

  Current state of welder device in its switch on-off cycle.

- Public float deviceCooldown :

  In every ‘deviceCooldown’ seconds welder object switches on/off.

- Private float timePassedAfterUse : Time passed after switching on/off welder object. Used to compare time with ‘deviceCooldown'



Methods :

- private void CheckPlayer () :

The control method which spawns raycast towards to 'targetVector' vector. If the raycast hits an object and if tag of that object is 'Player' then GetHurt method of PlayerHurt class is called for player object.

- Private void ActivateDevice () :

Firstly, 'targetVector' is determined according to 'targetAxis' value and 'deviceActive' flag is set to 'true'. Then LineRenderer object is enabled and starting position of the line renderer is set to current position of the welder object, ending position position of the line renderer is set to the raycast's hit point. Finally, 'timePassedAfterUse' value is set to zero.

```
targetVector = (targetAxis < 0) ? ((targetAxis == -1) ? Vector2.left : Vector2.down) : ((targetAxis == 1) ? Vector2.right : Vector2.up);
deviceActive = true;
lineRenderer.enabled = true;
RaycastHit2D hit = Physics2D.Raycast(transform.position,targetVector);
Debug.DrawLine(new Vector3(transform.position.x,transform.position.y-100,transform.position.z), hit.point);
lineRenderer.SetPosition(0, transform.position);
lineRenderer.SetPosition(1, hit.point);
timePassedAfterUse = 0;
```

- Private void DeactivateDevice () :

Sets 'deviceActive' as false, disables 'lineRenderer' and sets 'timePassedAfter' value to zero.

- Private void Update () :

In Update loop the flow above is implemented.

```
if(deviceActive)
{
    CheckPlayer();
}
if (!deviceActive && timePassedAfterUse >= deviceCooldown)
{
    ActivateDevice();
}

if(deviceActive && timePassedAfterUse >= deviceCooldown)
{
    DeactivateDevice();
}
```

4.2.8 Projectile Classes :

Any object which is spawned from a Weapon class is a projectile in this solution such as default bullets, dropable bombs and homming missiles. All these projectiles are inherited from Projectile class.

Projectile Class :

Base class for all projectile type classes. It is also a child class of PoolObject class because projectiles are used repeatedly and too many object creation and destruction can cause efficiency issues.

Fields :

- protected string effectiveTargetType :

  Tag of game object which can be hurt by this projectile such as ‘Player’ or ‘Enemy’.

- Public bool interruptedByPlatforms = false :

  If value of this flag is ‘true’, then when the projectile collides with platforms it is destroyed.

- Public float moveSpeed :

  X axis velocity multiplier of the projectile.

Methods :

- protected virtual void Go () :

  Moves the projectile object according to ‘moveSpeed’ multiplier in the right direction as default.

- Protected virtual void Go (Vector2 target) :

  Overload of the previous method. Moves the projectile object according to given direction and ‘moveSpeed’ multiplier.

DefaultAmmo Class :

Implementation for the most common projectiles which is used in ‘assault rifle’, ‘turret’ and ‘keykeeper’ objects. When they are reused by OnObjectReuse (Weapon) method of their parent PoolObject class. They determine which tag they will be used against by using ‘currentOwner’ of Weapon class.

Methods :

- public override void OnObjectReuse (Weapon weapon) :

  When an object is reused from the object pool this method is called. When DefaultAmmo object is reused, the method checks Weapon object for its ‘currentOwner’ field. If ‘currentOwner’ field is ‘Player’

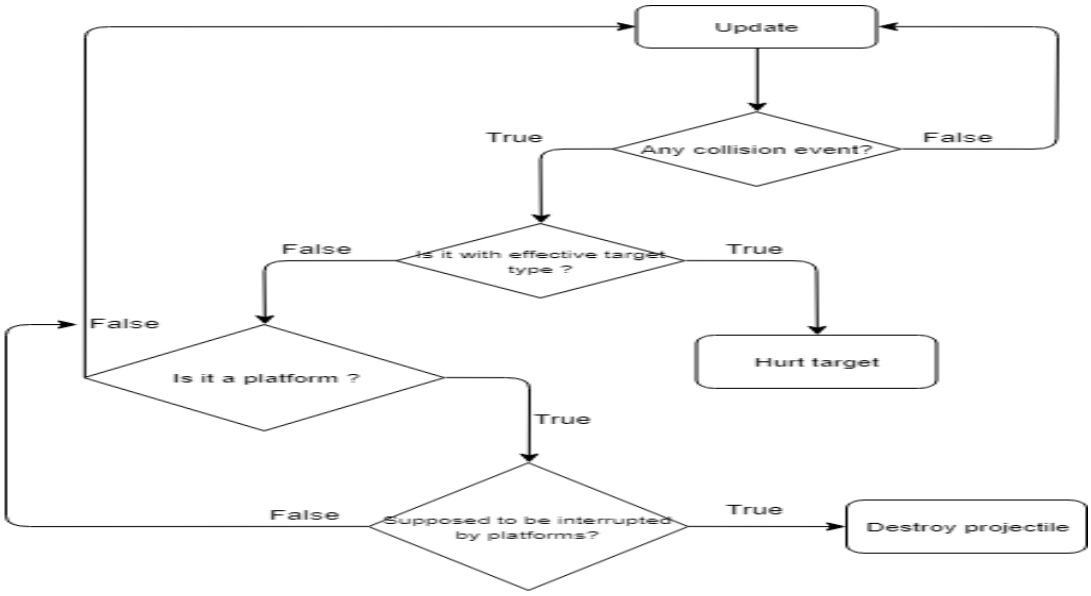
then it means the main character uses the weapon so ‘effectiveTargetType’ should be ‘Enemy. Otherwise, an enemy object is using the Weapon object so ‘effectiveTargetType’ becomes ‘Player’. After this setting, according to X axis of ‘lossyScale’ of the Weapon object, moving direction of the projectile is set.

```
float weaponSide = weapon.gameObject.transform.lossyScale.x;

if(Mathf.Sign(moveSpeed) != Mathf.Sign(weaponSide))
{
    moveSpeed *= -1;
}
```

- Private void OnTriggerEnter2D (Collider2D collider) :

  Built-in method of Unity Engine which checks ongoing collision events.



Flowchart: Collision checking for DefaultAmmo

When a collision event happens, the method checks if colliding object has ‘effectiveTargetType’ tag on it. If it is not the target type object, then the method checks if is it a platform. If it is a platform object then checks its ‘interruptableByPlatforms’ flag and it is ‘true’, the projectile object gets put back into the pool.

HommingMissile Class :

HommingMissile class is implementation of projectiles which tracks target for a specific period. It is child class of Projectile class which is also PoolObject.

Fields :

- private Transform target :

  Transform component of the target object.
- Public float rotateSpeed :

  How fast the missile will rotate when tracking the target.
- Private Rigidbody2D hommingMissileRigidbody :

  Rigidbody2D component of the missile object for moving.
- Private float timeAfterFiring :

  Time passed after the missile is fired. It is used for self-destruction after ‘attackCooldown’ seconds.
- Public float attackCooldown :

  Active duration for the missile object.
- Public GameObject explosionEffectObject :

  When the fired missile explodes this object is spawned to display explosion effect.

Methods :

- public override void OnObjectReuse (Weapon weapon) :

  Implementation is same with OnObjectReuse method of DefaultAmmo class. First, ‘effectiveTargetType’ is determined. Secondly, in addition to DefaultAmmo behaviour, reused missile object needs a game object with tag of ‘effectiveTargetType’ to track. To achieve that FindGameObjectWithTag method of GameObject built-int class is used.
- Private void void HandleTracking () :

  The method checks first if ‘target’ field is set or ‘null’ as an assurance to OnObjectReuse method. If it is null, then FindgameObjectWithTag method is called again with ‘effectiveTargetType’ field. After that, a Vector2 instance declared called ‘direction’ which represents direction vector between the target object and the missile. Then, ‘direction’ vector is normalized and cross producted with ‘up’ vector of Transform component of the missile. Z axis of the result vector is declared as ‘rotateAmount’ field which is a ‘float’ type. When this calculation is finished, ‘angularVelocity’ of the missile is set according to found ‘rotateAmount’ and ‘rotate speed’.

```
if(target == null)
{
    target = GameObject.FindGameObjectWithTag(effectiveTargetType).transform;
}
Vector2 direction = (Vector2)target.position - hommingMissileRigidBody.position;
direction.Normalize ();
float rotateAmount = Vector3.Cross (direction, transform.up).z;

hommingMissileRigidBody.angularVelocity = -rotateAmount * rotateSpeed;
hommingMissileRigidBody.velocity = transform.up * moveSpeed;
```

- Protected void FireControl () :

  See FireControl method of ShooterEnemy class.

- Private void DeathProcedure () :

  The method when the missile met its target. It spawns an ‘explosionEffeectObject’, then gets disabled for future use in the object pool.

- Private void OnTrigger2D (Collider2D collider) :

  See OnTrigger2D method of DefaultAmmo class. Homming missile objects does not control for platforms, they can pass through platforms.

DropableBomb Class :

Implementation class for bomb objects which are thrown from Flyer objects. They simply drop from air. It is child class of Projectile class which is also PoolObject.

Fields :

- private Rigidbody2D bombBody :

  Rigidbody2D component of the bomb object for moving.

- Public GameObject explosionEffectobject :

  When the bomb explodes this object is spawned to display explosion effect.

Methods :

- protected override void Go() :

  In default Go method in parent class, projectiles go towards the right. However, these bomb objects fall from the air so they move towards bottom. In the method MovePosition direction set to Vector2.down vector.

```
bombBody.MovePosition(bombBody.position + Vector2.down * Time.fixedDeltaTime * moveSpeed);
```

- public override void OnObjectReuse (Weapon weapon) :

  See OnObjectReuse method of DefaultAmmo class.

- Private void OnTriggerEnter2D (Collider2D collider) :

  See OnTriggerEnter2D method of DefaultAmmo class.

- Private void DeathProcedure () :

  See DeathProcedure method for HommingMissile class.

4.2.9 Weapon Classes :

Weapon classes are used to spawn Projectile objects through PoolManager class. Because of Projectile objects are seperate behaviours, one generic Weapon class is usually enough. However, some Weapon classes have different behaviours such as auto aiming.

Weapon Class :

Fields :

- public GameObject ammoPrefab :

  What kind of Projectile object is going to be spawned from Weapon object.

- Public int ammoCount :

  How many ‘ammoPrefab’ will be created in the object pool.

- Public string weaponName :

  Name of the weapon such as ‘assault rifle’ or ‘rocket launcher’.

- Public float damage :

  Damage value of the weapon. When a target gets hurt from this weapon ‘damage’ value is substracted from target’s health.

- Private string currentUser :

  Current user for the weapon. It can be tag of ‘Player’ or ‘Enemy’ as mentioned in OnObjectReuse method in DefaultAmmo class.

Methods :

- protected void DetermineOwner () :

  This method checks this game object is a child object of ‘player’ object in Unity Hierarchy. If it is not child of the ‘player’ object, then it means current user for player is an enemy object and ‘currentUser’ value is set to ‘Enemy’ string. Otherwise, ‘currentUser’ is set to ‘Player’ string.

```
currentUser = (transform.GetComponentInParent<PlayerInput>() == null) ? "Enemy" : "Player";
Debug.Log ("Owner determined for " + gameObject.name + " current user is " + currentUser);
```

- Protected virtual void Start () :

  Initialization method of Unity Engine. It creates a pool for ‘ammoPrefab’ in singleton PoolManager class with number of ‘ammoCount’. DetermineOwner method is also called in Start method.

- Public virtual void Fire () :

  This method is responsible for spawing Projectile objects. It simply calls ReuseObject method of PoolManager class for current ammo prefab.

- Public virtual void Fire (Quaternion rotation) :

  Overload for previous Fire method. This version is able to pass rotation input to change rotation of Projectile spawned.

Turret Class :

Implementation of ‘turret system’ object in the scene. Turrets are rifle like weapons which aims and fires to player object automatically. Turret system is a child class of Weapon class.

Fields :

- public float angleForSprite :

  Angle of sprite for the turret object is import for exact rotation display. It is ‘90’ degrees because of it is drawn as vertical.
- Protected Tranform targetTransform :

  Transform component of the target.
- Protected bool ableToFire = true :

  Current state of the device.
- Public float attackCooldown :

  Time period between attacks.
- Protected float timeAfterFiring :

  Time pass after each shooting. Used for fire control.
- Protected Quaternion pureRotation :

  Calculated rotation witout angle of the sprite.
- Private Animator animator :

  Animation controller for ‘turret system’ object.

Methods :

- protected virtual void RotateTurret () :

  First, a vector declared with horizontal and vertical axis difference between target object and turret object positions. After that, Atan2 method of Matf built in class is called which returns angle in radiant of tanjent that its value is vertical divided by horizontal. Then, radiant value is converted into normal angle value. Finally, result angle is set as Z axis value of current rotation of the turret system’s Z axis value.

```
float AngleRad = Mathf.Atan2( targetTransform.position.y - transform.position.y , targetTransform.position.x - transform.position.x);
float angle = (180 / Mathf.PI) * AngleRad;
transform.rotation = Quaternion.Euler(0, 0, angle - angleForSprite);
```

- Protected virtual void FireControl () :

  See FireControl method in ShooterEnemy class.

- Public override void Fire () :

  Animation controllers is triggered to play ‘fire’ animation. Then Fire method of parent Weapon class is called with value ‘pureRotation’.

- Protected override void Start () :

  Start method of the base Weapon class object is called. Also, ‘targetTransform’ is set to ‘player’ object from the game scene by passing PlayerInput singleton class’ ‘gameObject.transform’ value.

4.2.10 Reward Classes :

Reward classes represent any item drops from Reward Chest objects or Enemy objects as a reward for players such as coins, health or ammo.

DropMaster Class :

DropMaster class is singleton class which is responsible for droping reward objects. When an enemy dies or a reward chest is opened, their classes simply request a coin and random reward object.

Fields :

- private const POOL_SIZE = 50 :

  50 object will be created for each prefab in ‘dropableItemsPrefabs’ container.

- Public GameObject[] dropableItemsPrefabs :

  This field holds item prefabs which have chance to drop after an enemy killed or a reward chest opened such as coins, health or ammo.

Methods *:*

- private void CreatePools () :

  In this method if there are prefabs in ‘dropableItemsPrefabs’, pools are created for each prefab in the container with count of ‘POOL_SIZE’ variable.

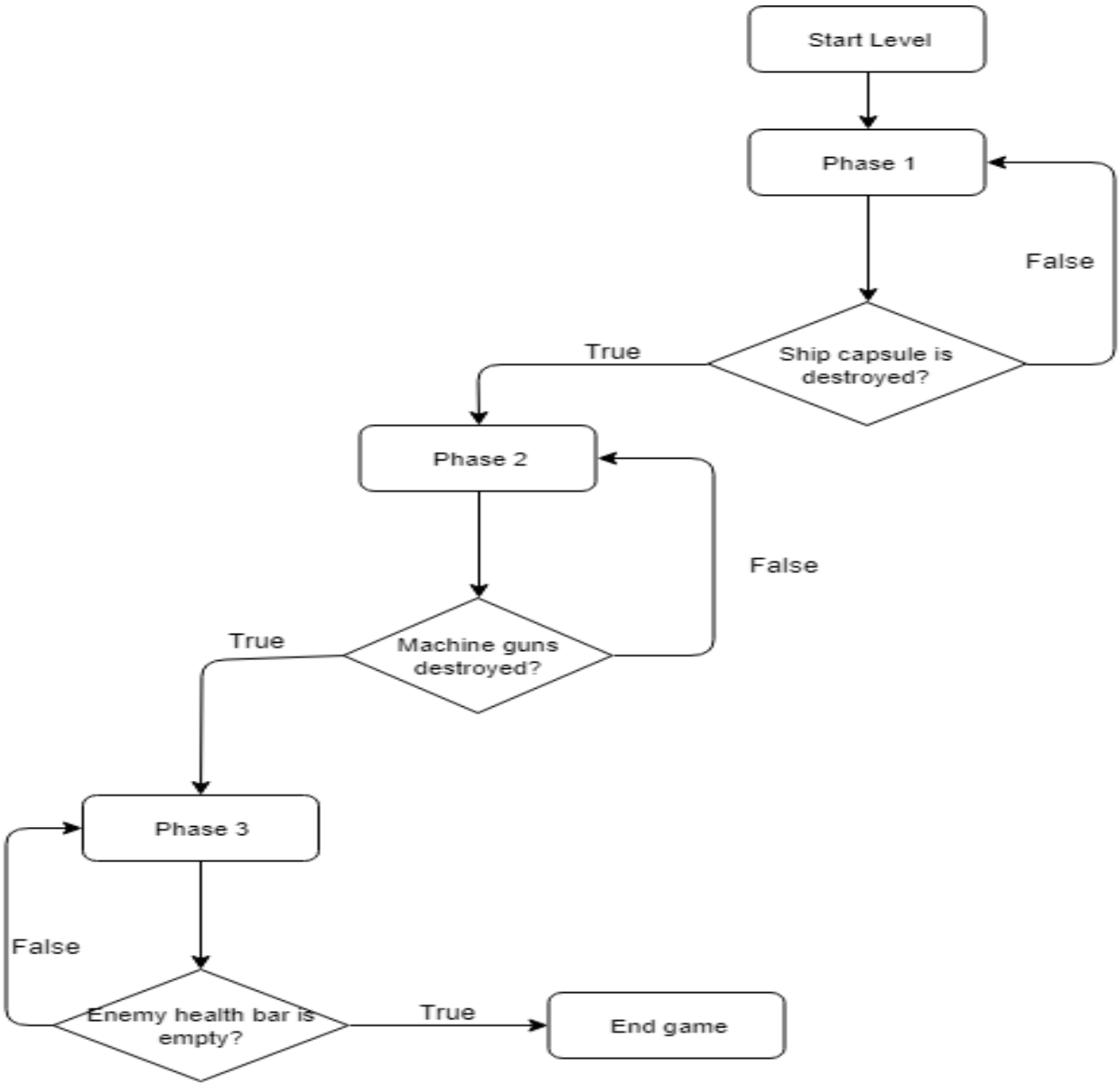
- Public void AskForDrop (Vector3 dropPosition) :

  This method randoms an index from ‘dropableItemsPrefabs’. After that, result index is called in ReuseObject method in PoolManager class with position of given ‘dropPosition’.

```
PoolManager.Instance.ReuseObject(
    dropableItemsPrefabs[Random.Range(0,dropableItemsPrefabs.Length -1)],
    dropPosition, Quaternion.identity);
```

4.3 Boss Scene

Boss Scene is where a powerful enemy waits for the main character. Main character has to pass 3 phases of the enemy to beat the game. Classes of the scene is reused in this scene. Only the boss object has different Weapon and Projectile objects similar to Level 1 Scene. The flowchart of boss fight is represented as following :



Boss Fight Phase

Boss phases in the level represent three different attack pattern of the boss In first phase, the boss throws missiles to the air. After few seconds the missiles comes back from botton or left side of the scene randomly. By the time, main character has to dodge the missiles and hit the capsule of space ship of the boss. When the capsule is destroyed, the second phase is started. In the second phase two machine guns appear from the bottom of the ship and behave like a turret system in Level 1 Scene. These machine guns can heat up as key keepers do in Level 1 Scene. When, they heated up main character has to attack them to destroy. After the machine guns destroyesd, the boss' space ship collapses and he starts to fly with a jetpack such as main character does. When he has same alignment with the main character he attacks with the current weapon that he holds. After last health bar drop is gone. Then the boss dies and the game ends.

4.3.1 Boss Scene Classes

Boss Scene contains following classes :

- BossStraightLauncher Class
- BossStraightRocket Class
- BossCageHurt Class
- BossMachinegunPit Class
- BossMG Class
- BossMGHurt Class

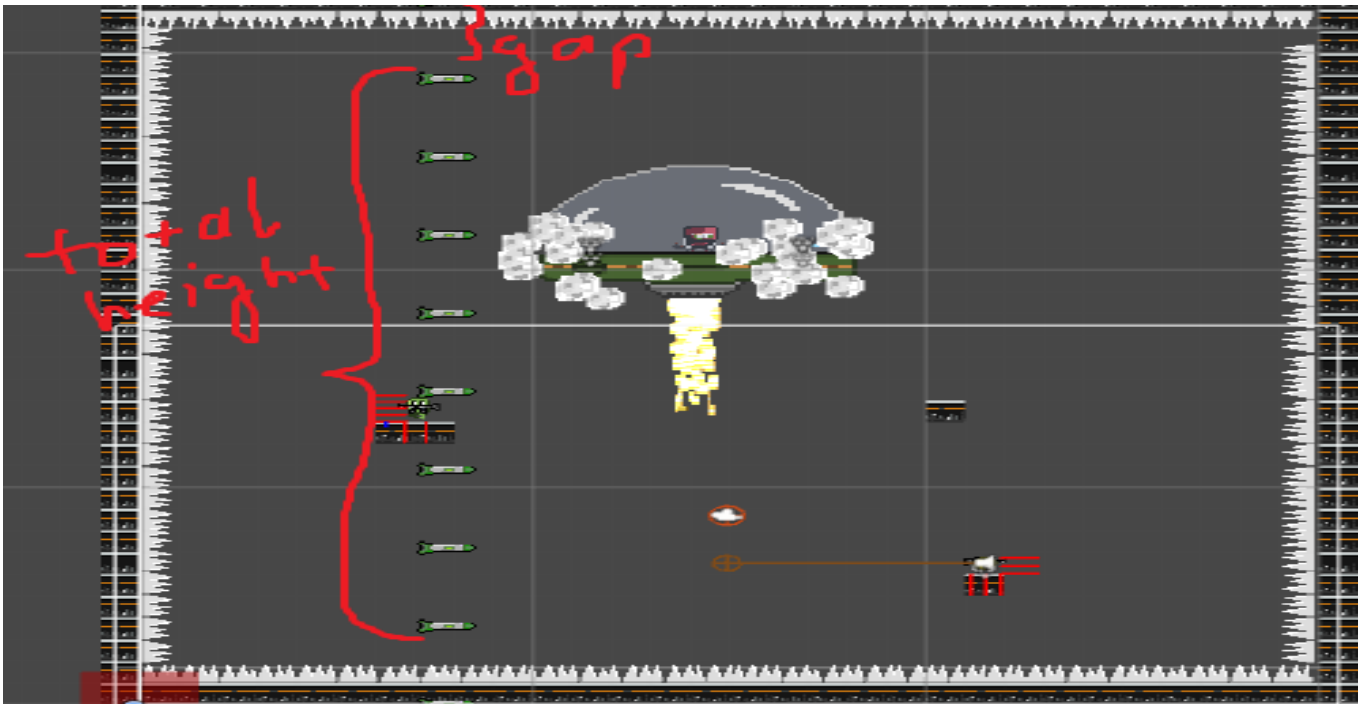
BossStraightLauncer Class :

This singleton class implements rocket launcher behavior of space ship of the boss. It works like a Weapon object. Creates ammo pools then fires them towards the main character in the first phase.

Fields :

- public float totalHeight :

  Desired height between first missile and last missile when the missiles come from right side



of the scene.

- Public float totalWidth :

  Desired width between first missile and last missile when the missiles come from bottom side of the scene. Horizontal version of ‘totalHeight’ figure.
- Public GameObject ammoPrefab :

  Ammo prefab is going to be created in object pool.
- Public int ammoCount :

  Ammo prefab pool size.
- Public bool isVertical :

  If ‘isVertical’ value is ‘true’, then the missiles comes from bottom side of the scene. Otherwise they come from left side.
- Public float fireCooldown :

  Time period between each missile attack.
- Public float eachPairOfRocketsSendingToAirTime :

  A missile attack consists of 5 pairs of rockets. Every pair has a time period between them this value defines it.
- Public float rocketsComingBackTime :

  When will the missiles comming back.
- Public float dangerGoneTime :

  When will be a missile attack is ended. This is the state where the boss is weak.
- Public bool phaseActive :

  Is this phase is active?
- Private List<Vector2> positionsToSpawnVertical :

  From which positions missiles enter the scene if they come from the bottom side.
- Private List<Vector2> positionsToSpawnHorizontal :

  From which positions missiles enter the scene if they come from the left side.

Methods *:*

- private void CalculatePositionsVertical () :

  Firstly, gap between each rockets in a missile attack is found by dividing ‘totalHeight’ by ‘ammoCount’. Then in a for loop these positions are added into ‘positionsToSpawnVertical’ lists by incremental with the gap value.

```
float verticalIncrease = totalHeight / ammoCount;
for (int i = 0; i < ammoCount; ++i)
{
    positionsToSpawnVertical.Add(new Vector2(0 + (i * verticalIncrease), 0));
}
```

- Private void CalculatePositionsHorizontal () :

  Firstly, gap between each rockets in a missle attack is found by dividing ‘totalWidth’ by ‘ammoCount’. Then in a for loop these positions are added into ‘positionsToSpawnHorizontal’ lists by incremental with the gap value.

- Private void FireRocketsVertical () :

  For each points in ‘positionsToSpawnVertical’ list, an ‘ammoPrefab’ object is reused at that position index.

- Private void FireRocketsHorizontal () :

  For each points in ‘positionsToSpawnHorizontal’ list, an ‘ammoPrefab’ object is reused at that position index.

- Private void FireRockets () :

  Firstly, Random class picks random integer between [0,1]. If randomized value is zero, then ‘isVertical’ value is set to ‘true’ which means a missile attack is going to come from bottom side. After that, FireRocketsVertical method is called. Otherwise, ‘isVertical’ value is set to’false’ which means a missile attack is going to come from left side. After that, FireRocketsHorizontal method is called.

- Private void HandleAutoFire() :

  Similar time based control which is implemented in FireControl method in ShooterEnemy. However, insted of calling direct Fire method. Firing of the missiles handled by animator to make synchronized animation and object creation at the same time. In the method, “firetrigger” of Animator trigger is called. When last frame of the animation occurs then FireAnim method is called.

- Public void FireAnim () :

  Starts RocketBarrage coroutine.

- Private IEnumerator RocketBarrage () :

  This coroutine handles timing between missile launches. Firstly, five pairs of rockets by are reused to top of the scene from spaceship sprite’s launcher spots to create the feeling of a missile rain.

Each pair of rockets waits ‘eachPairOfRocketsSendingToAirTime’ time. When five pairs are sent then the coroutine waits for ‘rocketComingBackTime’ seconds. Then FireRockets method is called. After that, the

coroutine waits for ‘dangerGoneTime’ that the missile attack passes through the top of level and disappears from sight. Then the launcher is ready to send a new wave of missiles.

BossStraightRocket Class :

Behavioral implementation of the missiles which is used by BossStraightLauncher class.

Fields :

- private RigidBody2D rocketBody :
  RigidBody2D component of a missile for movement.
- Public GameObject explosionEffectObject :
  Effect object which is going to be spawned after destruction of the missile.
- Private Vector2 direction :
  Direction vector for movement.
- Public int damage :
  Damage value which is dealt to target by a missile.

Methods :

- protected override void Go () :
  In the default implementation of Go method in base class, velocity direction is constant. In this override version the missile moves according to ‘direction value’.
- Public overide void OnObjectReuse () :
  In this reuse method, ‘direction’ vector is set according to ‘isVertical’ value of BossStraightLauncher class via singleton pattern. If ‘isVertical’ is currently set to ‘true’, then ‘direction’ vector is set as ‘Vector2.up’. If ‘isVertical’ is currently set to ‘false’, then ‘direction’ vector is set as ‘Vector2.right’.
- Private void OnTriggerEnter2D (Collider2D collider) :
  See OnTriggerEnter2D method of DropableBomb class.
- Private void DeathProcedure () :
  See DeathProcedure method of DropableBomb class.

BossCageHurt Class :

This class is a hurt class for the glass capsule which the boss stands inside. This class is inherited from EnemyHurt class and have health and destructible behavior. This class froms the first phase. When health of the glass capsule object becomes zero, then the first phase will be ended and the second phase will be started.

Methods *:*

Only difference between methods of this class and its parent class EnemyHurt class is override DeathProcedure method.

- Private override void DeathProcedure () :

In the default DeathProcedure method, when ‘currentHealth’ value of the object becomes zero. A reward drop is requested from DropMaster class and the object is disabled for future use. In addition to these functionality, when ‘currentHealth’ value of the object becomes zero, ‘phaseActive’ field of BossStraightLauncher class is set as ‘false’ to stop the first phase. Then, BossMachineGunPit class’ StartPhase class is called to start the second phase.

BossMachinegunPit Class :

This singleton class is for object container object which is a child of ‘bosspaceship’ object in Unity Hierarchy. The container is hidden at the first place after BossCageHurt class sends a notification about ending of the first phase. The machine gun pit reveals itself, and the guns start to shoot agains main character.

Fields :

- public float gunOpenningTimeFactor :

This is a multiplier factor to speed up revealing time of the guns at the start of the second phase.

- Private List<BossMG> guns :

BossMG objects which represent behaviour of weapons of the boss in the second phase.

- Public GameObject pitsLimiter :

This object is created to determine how far the machinegun pit object will reveal itself.

- Public bool phaseActive = false :

If ‘phaseActive’ value is set to ‘true’, the second phase is stared. This is the flag that used in loop of the second phase

- private bool openningDone = false :

The guns starts to shoot after reaveling of the machine gun pit object. This is the flag used for preventing the guns from starting before revealing.

- Private bool gunsActive = false :

This flag is used to contol the guns state such as FireControl methods of most auto shooting classes.

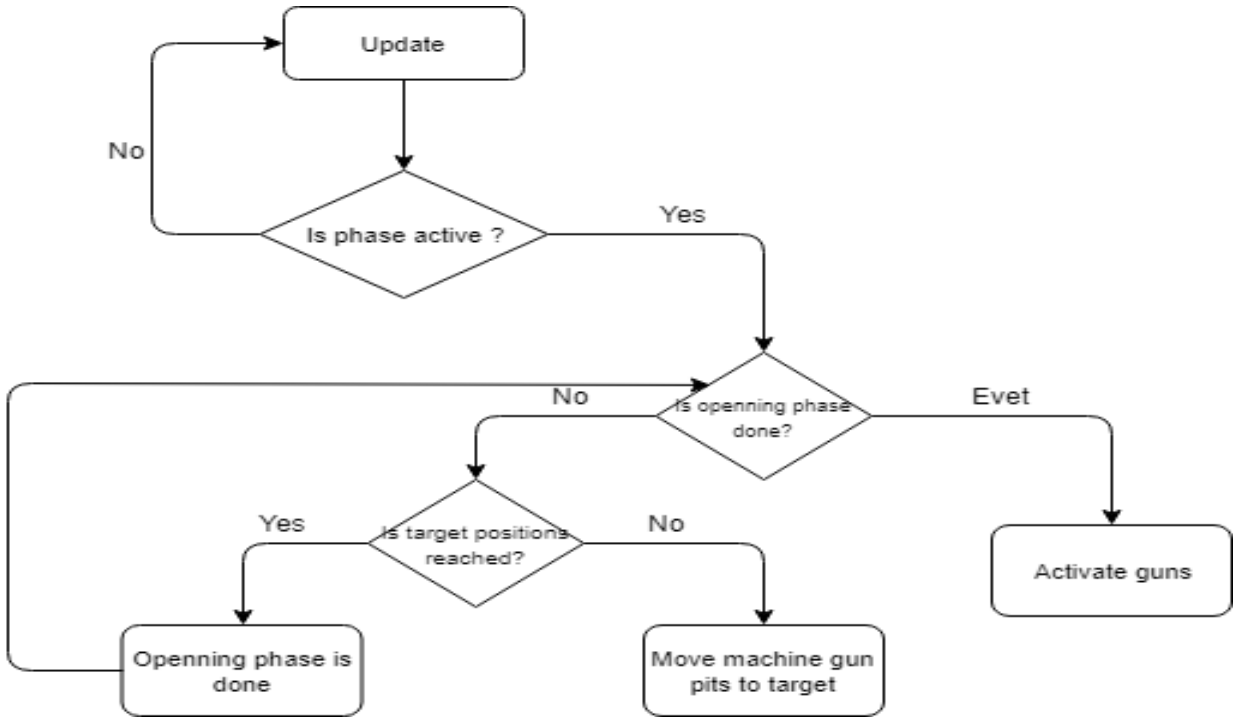
Methods *:*

- private void Start() :

In Start method, BossMG class instances are found by calling GetComponentsInChildren method. Found instances added to ‘guns’ list.

- Private void Update() :

In Update method openning phase of the machine gun pit object is handled. When the second phase is started the machine gun pit object starts to move position of the ‘pitsLimiter’ object according to ‘gunOpenningTimeFactor’ value. When positions is set, StartGuns coroutine is set.



- Private void ActivateGuns () :

  Sets ‘phaseActive’ fields of BossMG objects ‘true’. And “spin” animation is triggered from Animator component.

- Private IEnumarator StartGuns () :

  Sets ‘openningDone’ value to ‘true’ and wait for 2 seconds. Then calls ActivateGuns method.

BossMG Class :

BossMG objects have same implementation with KeykeeperMinigunScript and it is child class of Turret class. Only change between them is CheckWeaponHeat method which determines if turrets are heated up. In CheckWeaponHeat control in BossMG class, when turrets got heated up ‘spin’ animation of is stopped and when turrets got cooled down ‘spin’ animation is started again.

Fields :

See KeykeeperMinigun class.

Methods *:*

See KeykeeperMinigun class.

BossMGHurt Class :

As mentioned, in the second phase, main character have to destroy both turrets to pass the phase. BossMGHurt class is child class of HittableObject and works similar to any other hurt class.

Fields :

- public GameObject explosionEffectObject :

  In addition to ‘bloodSplash’ particles, there is also an game object which is for explosion effect. In DeathProcedure method this explosion effect objet is instantiated.

- For other fields, see BossCageHurt class.

- **Methods :**

  See methods of BossCageHurt class.

BossHealthBar Class :

This singleton class handles UI image which represent health bar of the boss.

Fields :

- public float currentHealth = 0 :

  Current health of the boss object.

- Public float maxHealth = 0 :

  Cummulative health of guns and the glass capsule. This max health is set in Start methods of BossMGHurt class and BossCageHurt classes.

Methods :

- public void DecreaseHealth (float amount) :

  This method updates the UI health bar image every time the glass capsule in the first phase or the guns in the second phase gets damage. According to ‘amount’ value, ‘currentHealth’ value is decreased. Then, local scale of image gets set to ‘currentHealth’ value divided by ‘maxHealth’ object so that the UI image seems like it is a decreasing bar.

Local scale = currentHealth / maxHealth = 1

Local scale = currentHealth / maxHealth = 0.7