Journal of Artificial Intelligence and Data Science (JAIDA)
e-ISSN: 2791-8335
https://dergipark.org.tr/pub/jaida

Vol. 3, No. 2, pp. 88-98, 2023
Research Article
Received: 06/09/2023 Accepted: 05/11/2023

Pixel Dungeon - Turn Based Game With Unity

Mehmet EROĞUL [1], Ömer Faruk CENİKLİ [2], Oğuzhan BİÇEN [3], Fidan NURİYEVA [4, *]

Abstract

Pixel Dungeon is a turn-based game developed using the C# programming language and the Unity game engine, targeting players who enjoy using strategic thinking skills to defeat their opponents. It is designed based on the Object-Oriented Programming concept and incorporates commonly used Design Patterns in game programming such as Observer, State, and Singleton. The objective of the game is to use the selected units (game characters) to battle against enemy units and emerge victorious, aiming to achieve the highest possible score. As we defeat the encountered enemies, we earn points and face new opponents. The game concludes when we lose all of our units. Each unit in the game possesses unique abilities (ranged attack, melee attack, healing, buffing or debuffing another unit, poisoning, etc.). Each character takes turns to make their moves and strategically utilizes their abilities to gain an advantage during the gameplay. This study serves as an example of how turn-based games can be designed using the Unity game engine. It not only provides an enjoyable gaming experience but also enhances players' strategic decision-making skills.

Keywords: Game development; object oriented programming; turn based game; unity.

1. Introduction

The gaming industry has witnessed remarkable growth in recent years, and Unity has emerged as one of the leading game development engines. Game development is a rapidly evolving field, and the use of the C# programming language in combination with the Unity game engine has become a predominant choice among developers. Unity offers several advantages to game developers, making it a popular choice in the industry [1, 2]. Unity is a powerful and versatile game development engine that has gained immense popularity among developers worldwide. Unity is a cross-platform game development engine that enables developers to create games for various platforms, including PC, consoles, mobile devices, and etc. It boasts an intuitive user interface and a vast array of tools, making it accessible to developers of all skill levels. Unity supports multiple programming languages like C#, JavaScript, with C# being the most widely used [3, 4]. C# has emerged as the scripting language of choice for Unity development. It strikes a balance between performance and ease of use. Developers appreciate C#'s versatility and strong support within the Unity ecosystem. C# and Unity have made significant inroads into various game genres, including action, adventure, simulation, strategy, and more. Beyond entertainment, Unity and C# have been employed in developing serious games, educational simulations, and applications in augmented and virtual reality.

There are numerous games developed using C# and Unity. Hollow Knight, Cuphead, Oxenfree, Inside, Cities: Skylines, Monument Valley, Crossy Road, Kerbal Space Program, Pillars of Eternity are some of them.

Pixel Dungeon is a turn-based game project. It is developed using the C# programming language and the Unity game engine. It is designed to challenge players' strategic thinking skills as they strive to defeat their opponents.

An earlier version of this paper was presented at the ICADA 2023 Conference and was published in its Abstract Book (Title of the conference paper: "Unity ile Sıra Tabanlı Oyun").

The goal of the game is to use the selected units (game characters) to engage in battles against enemy units and emerge victorious to achieve the highest possible score. As players defeat their adversaries, they earn points

*Corresponding author

MEHMET EROĞUL; Dokuz Eylul University, Faculty of Science, Department of Computer Science, Türkiye; e-mail: mehmet_erogul@icloud.com;

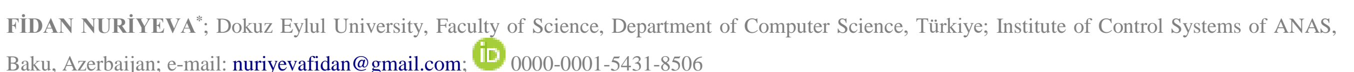
0009-0005-3352-0365

ÖMER FARUK CENİKLİ; Dokuz Eylul University, Faculty of Science, Department of Computer Science, Türkiye; e-mail: omerfarukceniklii@gmail.com;

0009-0008-3607-7030

OĞUZHAN BİÇEN; Dokuz Eylul University, Faculty of Science, Department of Computer Science, Türkiye; e-mail: oguzhanbicen.b@gmail.com;

0009-0002-2345-0015

FİDAN NURİYEVA*; Dokuz Eylul University, Faculty of Science, Department of Computer Science, Türkiye; Institute of Control Systems of ANAS, Baku, Azerbaijan; e-mail: nuriyevafidan@gmail.com;

0000-0001-5431-8506

and encounter increasingly challenging new enemies. However, if all of their units are defeated, the game comes to an end.

In the game, there are four units controlled by the player and four units controlled by the computer. The turn-based system in the game manages the order of actions for each unit based on their individual speeds. The units of both the player and the computer are placed in a queue according to their speeds. The unit with the highest speed will be the first one to perform its action, while the unit with the lowest speed will be the last one to act. Each unit takes its turn to perform its action and is then removed from the queue. This process continues until there are no more units left in the queue. After all units have taken their turns, a new round begins, and the turn-based system creates a new queue for the upcoming actions. The game proceeds with the next unit in the queue, and the cycle continues.

This turn-based mechanism adds a strategic element to the game, as players need to carefully plan their unit's actions and consider the order in which they will act to gain an advantage over the computer-controlled units.

By utilizing the turn system and considering the speed of each unit, players can make tactical decisions to optimize their chances of victory in battles against the computer-controlled units.

Figure 1. *The game's screenshot* [5, 6]

In the game, each unit has different abilities or actions it can perform (ranged attack, melee attack, healing, buffing or debuffing another unit, poisoning, etc.). The usage of abilities can be restricted based on the team or position of the characters. For example, the healing ability can only be applied to characters within the same team, while the attack ability can only be used against characters in the opposing team. The Cavalry unit can attack any position of the opponent from any position on the player's side, whereas the Knight unit can only attack one of the opponent's front two positions when it is in one of the player's front two positions.

Additionally, units have status attributes such as health points, attack, and defense. These attributes determine how much damage the unit will deal to the opponent, how much damage it will ignore when attacked, or how much damage it needs to take to be defeated. Some units' buff and debuff abilities can affect these status attributes.

2. How to Play

To play Pixel Dungeon, you only need to use the left mouse button. The game starts with the player selecting four out of seven characters, each having different abilities. As the game begins, the player encounters waves of enemies, each consisting of four enemy units, which progressively increase in difficulty.

Being a turn-based game, both the player's units and the enemy units take their actions one by one. A unit must complete its action before another unit can perform its action. If it is the player's turn, they can select any action from their current unit's ability set, and choose a valid position to execute the action by clicking the left mouse button. The player then initiates the selected action.

When it is the enemy's turn, the computer randomly attacks any of the player's units. The player must strategically plan their moves to defeat the enemy waves and achieve the highest score possible.

In the game, the player earns points by destroying any of the enemy units. When all enemy units are destroyed, the current score is doubled, and the next enemy wave appears. If the player loses all of their units, the game ends. The objective is for the player to keep their units alive as much as possible and achieve the highest score.

Figure 2. *Sample ability set for the archer unit* [7, 8]

Figure 3. *Display image of suitable positions for the selected skill* [5]

3. Game Development Environment in Unity

Unity is a cross-platform game engine that supports scripting with C#. It allows developers to process 2D and 3D graphics in real-time, play audio and animations, and easily control events such as collision detection. Unity provides an environment where we can easily manage various aspects of our game. Additionally, Unity includes a physics engine, enabling us to simulate physics in our game environment closely resembling real-world physics. This allows for a more realistic and immersive gaming experience.

In Unity, our game is composed of scenes. Scenes are structures that contain the objects of the game, such as cameras, lights, 3D characters or environment models, canvases, 2D images, and more. The objects present in the current scene are visible in Unity's Hierarchy panel. Each object has its own unique components that determine its behavior. These components are visible in Unity's Inspector panel when we click on an object in the Hierarchy.

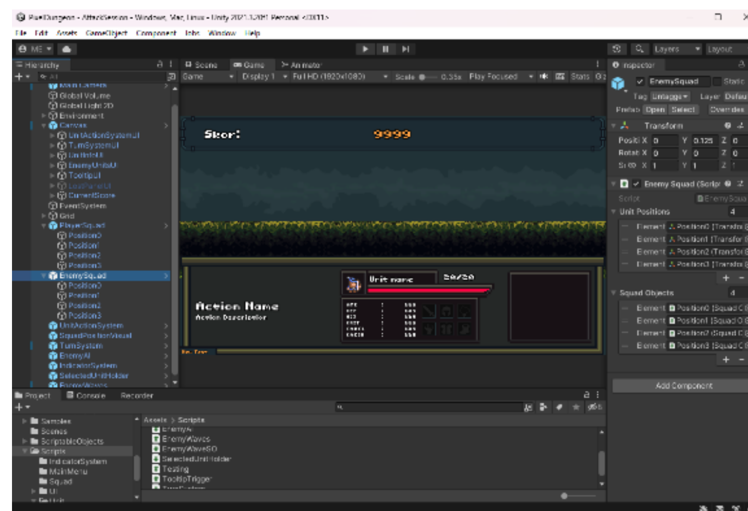


Figure 4. *Unity interface*

By default, a transform component is assigned to every object added to the scene. This component determines the object's position in the scene, its orientation, and its size.

When we want to play an audio file, we need to add the AudioSource component to the respective object. When we want to play an animation, we use the Animator component. For collision detection, we add the Collider component, and for applying physics, we add the Rigidbody component to the desired object. Unity provides many built-in components similar to these.

Additionally, we can create custom scripts and attach them as components to objects, allowing us to easily interact with other components of the object. By creating a custom script, we can play an audio clip through the AudioSource component of an object, move an object by changing the position values of its Transform component when a key is pressed, and decide which animation to play through the Animator component.

In summary, Unity offers a wide range of components that can be added to objects, and we can create custom scripts to control and manipulate these components efficiently. This flexibility allows for powerful and dynamic interactions within the game or application.

3.1. Pixel Dungeon scene structure

Scenes can be used to create the main menu or levels of the game. Pixel Dungeon consists of three scenes: the main menu, unit selection, and battle stages.

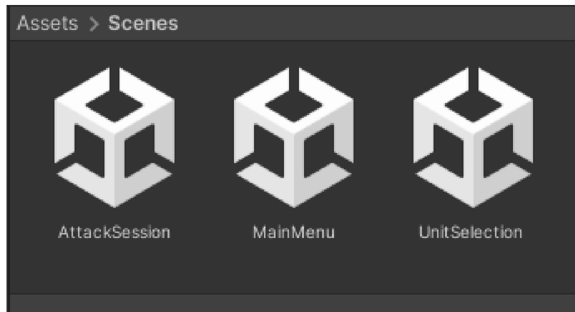


Figure 5. *Scenes of the game*

Figure 6. *Screenshot of the game's main menu scene*

When the player presses the "Play" button in the menu, the game transitions to the unit selection scene. In this scene, the player can choose four units with different abilities. After selecting the units, the player can click the "Start" button to transition to the battle scene.

Figure 7. *Screenshot of the game's unit selection scene*

When the battle scene is entered, the game starts. At this stage, pre-defined enemies appear in waves in front of the player. If the player loses the game, the losing panel, which is inactive on the canvas of the current scene, becomes active.

Figure 8. *Screenshot of the game's battle scene*

Figure 9. *Screenshot of the game's losing panel*

3.2. Objects and scripts in scenes

3.2.1. Main menu scene

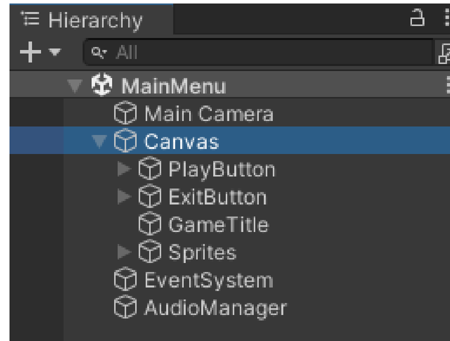


Figure 10. *Objects in the main menu scene*

In the main menu scene, there is one camera that displays a canvas containing text, images, and buttons to start the game or exit the game. In Unity, cameras are objects through which players view the world. Canvases, on the other hand, are structures that contain user interface elements.

In the scene, the canvas object has a script component named "MainMenu" attached to it. This script contains the functions "StartGame()" and "QuitGame()". These functions are called through the OnClick events of the "Play" and "Quit" buttons, respectively, enabling the player to proceed to the next scene or exit the game.

Additionally, there is an AudioManager object in the scene, which has an AudioSource component to play the background music of the game. The object is assigned a script with the same name. With this script, we use Unity's DontDestroyOnLoad() function to carry the object to the next scene. This ensures that the AudioManager persists across scene changes, allowing the background music to continue playing seamlessly as the player moves between scenes.

In Unity, data is lost when scenes are loaded. The DontDestroyOnLoad() function allows us to achieve scene-to-scene data persistence and transfer the background music to another scene without interruption. This ensures that the data associated with the object remains intact and carries over seamlessly as the player navigates between different scenes.

3.2.2. Unit selection scene

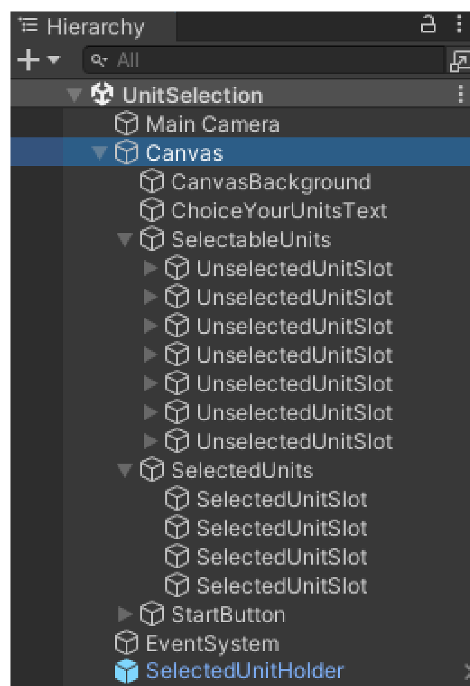


Figure 11. *Objects in the unit selection scene.*

In the unit selection scene, there is one camera and one canvas. The canvas has a script called "UnitSelectionManagerUI" attached to it, and this script allows the player to choose units and move them to slots. Within the canvas, there are seven slot objects (UnselectedUnitSlot) to hold unselected units and four slot objects (SelectedUnitSlot) to hold selected units. These objects have scripts with the same name, and these scripts store information about whether the slot is empty or which unit is assigned to that slot.

At the beginning, units are located under the UnselectedUnitSlot objects and they have a script called "UnitSelector". This script stores information about which slot the unit is assigned to and the unit's Prefab data. In Unity, Prefabs allow objects to be stored and reused, enabling efficient management and instantiation of objects throughout the game.

In the scene, there is also an object named "SelectedUnitHolder" which is assigned a script with the same name. This object is used to transfer the selected units to the Battle scene when the "Start" button is pressed. It collects the Prefab data from the UnitSelector scripts of the units assigned to the SelectedUnitSlot objects and stores them in a four-indexed Transform array. Then, it uses the DontDestroyOnLoad method to carry the data of the selected units to the next scene, ensuring their persistence and availability in the Battle scene.

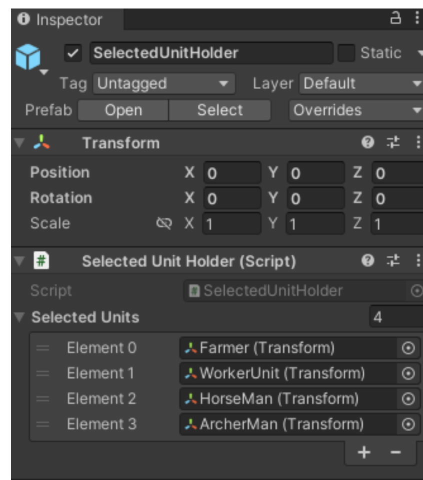


Figure 12. *The components of the "SelectedUnitHolder"*

3.2.3. Battle scene

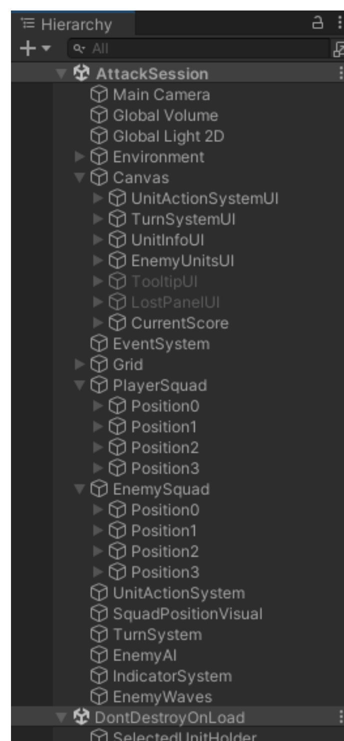


Figure 13. *Objects in the battle scene.*

In the battle scene, there is an object named "TurnSystem" which controls the entire game. When the scene is loaded, it receives the units sent from the "SelectedUnitHolder" object in the unit selection scene and passes them to the "PlayerSquad" object. The "PlayerSquad" object stores the position data where the units will be instantiated.

Figure 14. *The components of the "PlayerSquad"*

The "EnemySquad" object holds the position data for enemy units. When the "TurnSystem" is loaded or when the game transitions to the next enemy wave, it retrieves the relevant enemy units from the "EnemyWaves" object and sends them to the "EnemySquad" object. The "EnemyWaves" object stores enemy waves using Unity's ScriptableObject structure. ScriptableObject is a data container provided by Unity that can be used to save large amounts of data. In this example, ScriptableObjects are used to hold four enemy units each and are stored in an array of ScriptableObjects.

The TurnSystem, after sending the player and enemy units to the PlayerSquad and EnemySquad objects, respectively, organizes all units into a queue based on their speeds, creating a sequence in decreasing order of speed. It selects the unit with the highest speed, and if the unit belongs to the player, it assigns that unit to the UnitActionSystem object using the SetSelectedUnit() function of the object's UnitActionSystem script. If the next unit in the sequence is an enemy unit, it is sent to the EnemyAI object.

Each unit has a script called "Unit" associated with it. This script holds various data related to each unit. Additionally, units have scripts named "StatSystem" and "HealthSystem". These scripts store data such as the unit's attack, defense, and health points.

When a unit is assigned to the UnitActionSystem object, buttons are dynamically created on the canvas through the UnitActionSystemUI object for the abilities that the unit possesses. It also allows updating descriptions related to the player unit's abilities on the canvas. The OnClick event of the buttons calls the SetSelectedAction() function in the UnitActionSystem object, changing the selected action.

Specific command files such as MoveAction for the walking action and AttackAction for the attack action are created, each derived from the BaseAction class for every action. Valid positions are defined for each action. When the player clicks on the buttons, the valid positions for that action are visualized through the SquadPositionVisual object.

If the position clicked by the player is valid, the UnitActionSystem initiates the TakeAction() function of the selected unit's selected action, utilizing the data of the selected unit and action. After the action is completed, the TurnSystem comes into play, and the turn is passed to the next unit.

The IndicatorSystem object is used to create damage or healing numbers on the relevant unit whenever any unit takes damage or gets healed, providing feedback to the player.

The EnemyAI object is responsible for performing the attack action of the enemy unit it receives, targeting any of the player's units.

The UnitInfoUI object within the canvas updates data such as the selected player unit's attack, defense, and health points on the canvas. The EnemyUnityUI object is used to display and update enemy unit health on the canvas. The TooltipUI object displays data such as statuses, buffs, and debuffs of units when the mouse hovers over them.

The LostPanelUI object displays the score data on the screen if the player loses the game and contains a button to restart the game.

4. Used Design Patterns

Design patterns are standardized solutions created for commonly encountered problems in software development. Three design patterns were used while developing Pixel Dungeon: Singleton, Observer, and State design patterns.

4.1 Singleton design pattern

The Singleton design pattern ensures that only one instance of a class is created. The concept of an instance refers to allocating memory in the computer's memory for an object derived from a class. When we want to have only a single instance of an object, we use the Singleton design pattern.

In Figure 15, an example of a singleton used in Pixel Dungeon can be seen. A static instance is defined for the EnemyAI object. If there is no existing instance, assignment to the current instance is done within Unity's provided Awake function. If an instance has been assigned before, the current object is destroyed.

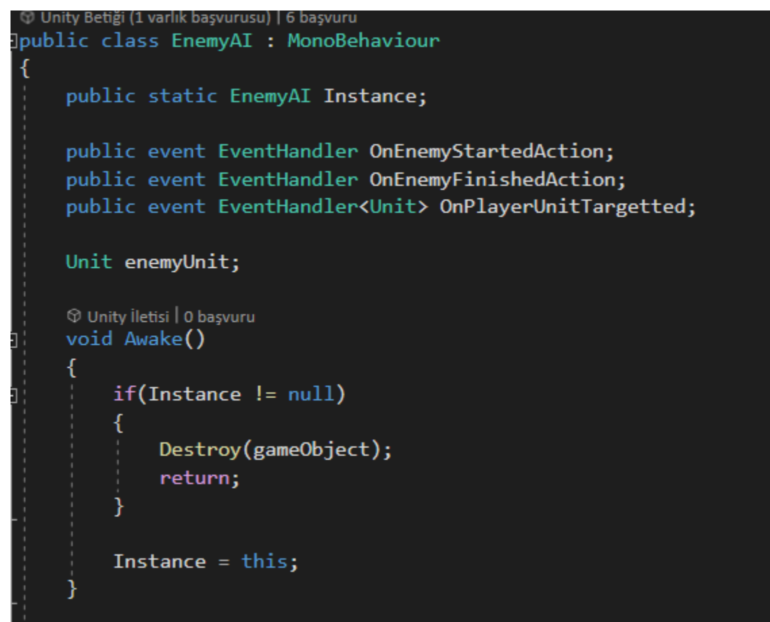


Figure 15. *Image of the EnemyAI object's EnemyAI script.*

When we create a script, we also create a class at the same time. By default, Unity inherits this class from the MonoBehaviour class. This allows us to attach that script as a component to an object. Additionally, the MonoBehaviour class provides us with various functions within the script that we can use, such as Awake, Start, LateUpdate, Update, FixedUpdate, OnEnable, and OnDestroy.

The Start and Update functions are automatically created in the class. They are called once when a scene is loaded. The only difference between the Awake and Start functions is that the Awake function is called before the Start function.

The Update function, on the other hand, works throughout the time our game is running and is called as many times as the computer's hardware allows (FPS - Frames Per Second).

The LateUpdate and FixedUpdate functions are similar to the Update function but have a different execution order, which is predetermined by Unity.

4.2 Observer design pattern

The Observer design pattern is used to easily notify all relevant objects when the state of an object changes. Its greatest benefit is minimizing the dependency between classes. The Observer design pattern can be implemented using C#'s event and delegate structures. In Pixel Dungeon, the ready-made delegate structure of C#, EventHandler, is used. Below is an example of the Observer design pattern used in Pixel Dungeon.

```
Unity Betiği (1 varlık başvurusu) | 25 başvuru
public class UnitActionSystem : MonoBehaviour
{
    public static UnitActionSystem Instance;

    public event EventHandler OnSelectedUnitChanged;
    public event EventHandler OnSelectedActionChanged;
    public event EventHandler<bool> OnBusyChanged;
    public event EventHandler OnActionStarted;
    public event EventHandler OnActionComplete;
```

Figure 16. *Events defined in the UnitActionSystem command line.*

In Figure 16, it can be observed that in the UnitActionSystem script, an event named OnSelectedActionChanged is defined to be used when the player's selected ability is changed.

In Figure 17, it shows how the event is invoked. The SetSelectedAction function is assigned to the OnClick events of buttons in the player's current unit's ability set. When the Invoke() function is called, other classes that have subscribed to this class's event will be notified about which ability the player has selected.

```
3 başvuru
public void SetSelectedAction(BaseAction baseAction)
{
    selectedAction = baseAction;

    OnSelectedActionChanged?.Invoke(this, EventArgs.Empty);
}
```

Figure 17. *The invocation method of the OnSelectedActionChanged event.*

```
public class UnitActionSystemUI : MonoBehaviour
{
    void Start()
    {
        tooltipContainer.SetActive(false);

        UnitActionSystem.Instance.OnSelectedUnitChanged += UnitActionSystem_OnUnitSelected;
        UnitActionSystem.Instance.OnSelectedActionChanged += UnitActionSystem_OnSelectedActionChanged;
        UnitActionSystem.Instance.OnBusyChanged += UnitActionSystem_OnBusyChanged;
        TurnSystem.Instance.OnUnitChanged += TurnSystem_OnUnitChanged;
        EnemyAI.Instance.OnEnemyStartedAction += EnemyAI_OnEnemyStartedAction;
```

Figure 18. *Subscribe to the OnSelectedActionChanged event from the UnityActionSystemUI script.*

In Figure 18, the way UnitActionSystemUI class subscribes to the OnSelectedActionChanged event of the UnitActionSystem class is shown. Inside the class, the function "UnitActionSystem_OnSelectedActionChanged" is defined, and whenever the event is called, this function will be called along with the event.

```
1 başvuru
private void UnitActionSystem_OnSelectedActionChanged(object sender, EventArgs e)
{
    UpdateSelectedVisual();
    UpdateActionTooltip(UnitActionSystem.Instance.GetSelectedAction());
}
```

Figure 19. *The UnitActionSystem_OnSelectedActionChanged function*

The "UnitActionSystem_OnSelectedActionChanged" function is used to change the background color of the selected ability and update the action's description text on the canvas based on the player's selection.

4.3 State design pattern

The State design pattern is a design pattern that allows an object to change its behavior when its internal state changes. It is closely related to finite state machines. Below is an example of the State design pattern used in Pixel Dungeon.

In Figure 20, it can be seen that in the AttackAction class, there is an enum named "State" created for states, and a variable named "state" is defined from this enum. Additionally, a counter named "stateTimer" is defined to determine the duration of each state.

```
public class AttackAction : BaseAction
{
    public event EventHandler OnAttack;
    public static event EventHandler OnAnyUnitStartedAttack;
    [SerializeField] AudioClip attackSound;

    10 başvuru
    private enum State
    {
        Aiming,
        Hitting,
        Cooloff,
    }

    private State state;
    private float stateTimer;
```

Figure 20. *Enumeration definition for states in the AttackAction class*

```
void Update()
{
    if(!isActive)
    {
        return;
    }

    stateTimer -= Time.deltaTime;

    float lerpSpeed = 8f;
    switch(state)
    {
        case State.Aiming:
            transform.position = Vector3.Lerp(transform.position, targetPosition, lerpSpeed * Time.deltaTime);
            break;
        case State.Hitting:
            if(canHitDamage)
            {
                Hit();
                canHitDamage = false;
            }
            break;
        case State.Cooloff:
            transform.position = Vector3.Lerp(transform.position, unitPosition, lerpSpeed * Time.deltaTime);
            break;
    }

    if(stateTimer <= 0f)
    {
        NextState();
    }
}
```

Figure 21. *Control of defined states within a switch.*

In Figure 21, it can be observed that the control of states is done using a switch inside the Update function. Using a switch is the simplest method to control states. When units perform an attack action, the AttackAction class comes into play. In the "Aiming" state, the attacking unit moves towards the position of the target unit.

In the "Hitting" state, the attack action is performed. In the "Cooldown" state, the unit returns to its previous position.

In the Update function, the value of the stateTimer counter variable is decreased over time using Unity's Time class. At the end of the function, if the time reaches zero, the NextState() function is called to transition to the next states.

```
1 başvuru
void NextState()
{
    switch(state)
    {
        case State.Aiming:
            state = State.Hitting;
            unit.GetAudioSource().PlayOneShot(attackSound);
            float hittingStateTime = 1f;
            stateTimer = hittingStateTime;
            break;
        case State.Hitting:
            state = State.Cooloff;
            float cooloffStateTime = 0.75f;
            stateTimer = cooloffStateTime;
            break;
        case State.Cooloff:
            ActionComplete();
            unit.SendSpriteBack();
            break;
    }
}
```

Figure 22. Control of state transitions in the NextState function.

In Figure 22, the content of the NextState function can be seen. When this function is called, it transitions to the next state and sets the value of the stateTimer counter. Additionally, it plays the attack sound effect of the unit and notifies relevant classes that the action is completed.

5. Conclusion

In this study, a game has been developed using the C# programming language and the Unity game engine. It caters to a niche of players who appreciate the intellectual challenge of strategic thinking and tactics. By incorporating the principles of Object-Oriented Programming and implementing well-established Design Patterns such as Observer, State, and Singleton, the game demonstrates a commitment to sound software engineering practices.

The core objective of proposed game project is to engage players in a battle of wits, utilizing a diverse array of in-game characters with unique abilities. This strategic depth adds a layer of complexity and intrigue to the gameplay, making each move a critical decision that can influence the outcome of the game. As players progress through the game, accumulating points and facing increasingly formidable opponents, they are continuously challenged to adapt their strategies and make the best use of their character's abilities. "Pixel Dungeon" not only offers an enjoyable gaming experience but also serves as an effective tool for enhancing players' strategic decision-making skills.

This study exemplifies how the Unity game engine can be harnessed to create turn-based games that not only entertain but also provide a platform for players to sharpen their cognitive and strategic abilities. "Pixel Dungeon" is a testament to the creative possibilities that arise from the fusion of innovative game design and state-of-the-art development technologies.

References

[1] J. K. Haas, “A history of the unity game engine”, Diss. Worcester Polytechnic Institute, 2014.

[2] P. B. Addison, J. Manning, and T. Nugent, “Unity Game Development Cookbook: Essentials for Every Game”, O'Reilly Media, 2019.

[3] H. Ferrone, “Learning C# by Developing Games with Unity: Get to grips with coding in C# and build simple 3D games in Unity 2022 from the ground up”, 7th Edition.

[4] S. L. Kim, , H. J. Suk, , J. H. Kang, J. M.Jung, T. H. Laine and Westlin, J., “Using Unity 3D to facilitate mobile augmented reality game development”. In 2014 IEEE World Forum on Internet of Things (WF-IoT), pp. 21-26, IEEE, 2014.

[5] LYASeeK, MiniFolks bundle. https://lyaseek.itch.io/

[6] Szadi art, Platformer Fantasy SET1. https://szadiart.itch.io/platformer-fantasy-set1

[7] M. Tohami, Pixel Art GUI Elements. https://mounirtohami.itch.io/pixel-art-gui-elements

[8] 7Soul, 7Soul's RPG Graphics - Icons. https://7soul.itch.io/7souls-rpg-graphics-pack-1-icons