AKDENİZ UNIVERSITY

FACULTY OF ENGINEERING
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

CSE 492
SENIOR DESIGN PROJECT II
INTERMEDIATE REPORT

Supervisor: Asst. Prof. MUSTAFA BERKAY YILMAZ

May 14, 2023

Selim Aybars Duran - 20190808019
Bengisu Şahin - 20180808052

Contents

1 Low Fidelity / High Fidelity Mockups

1.1 Low Fidelity Mockups

These are the low-fidelity mockups images:

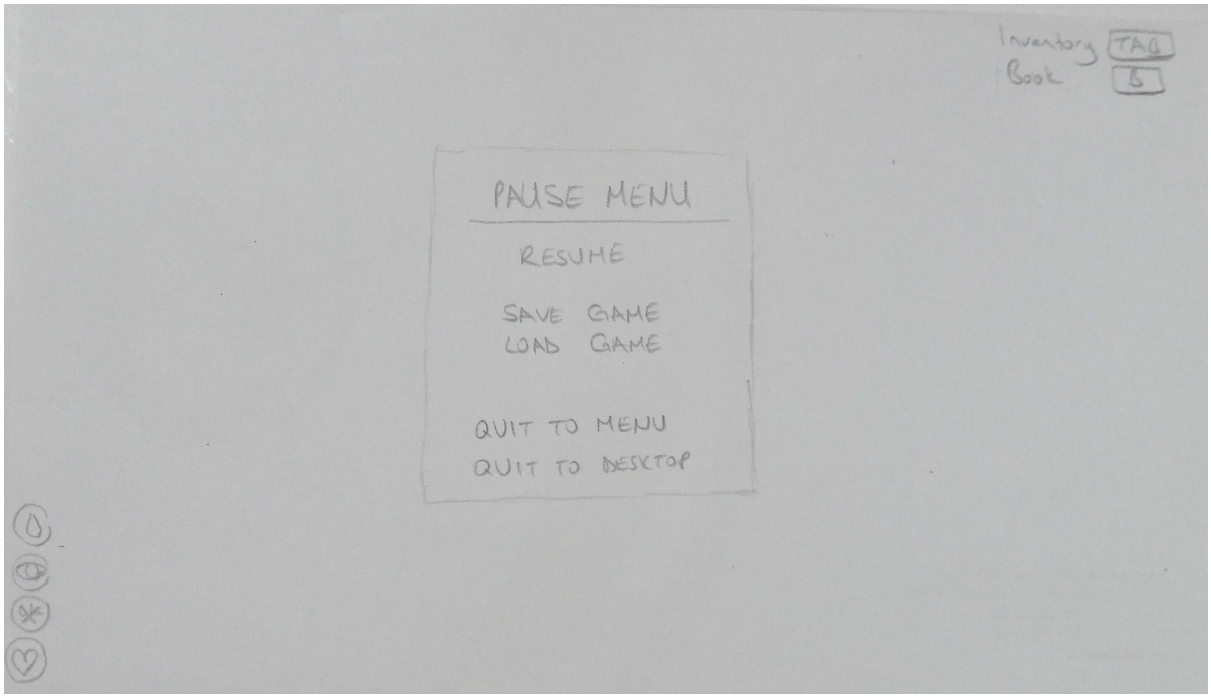


Figure 1: Pause Menu

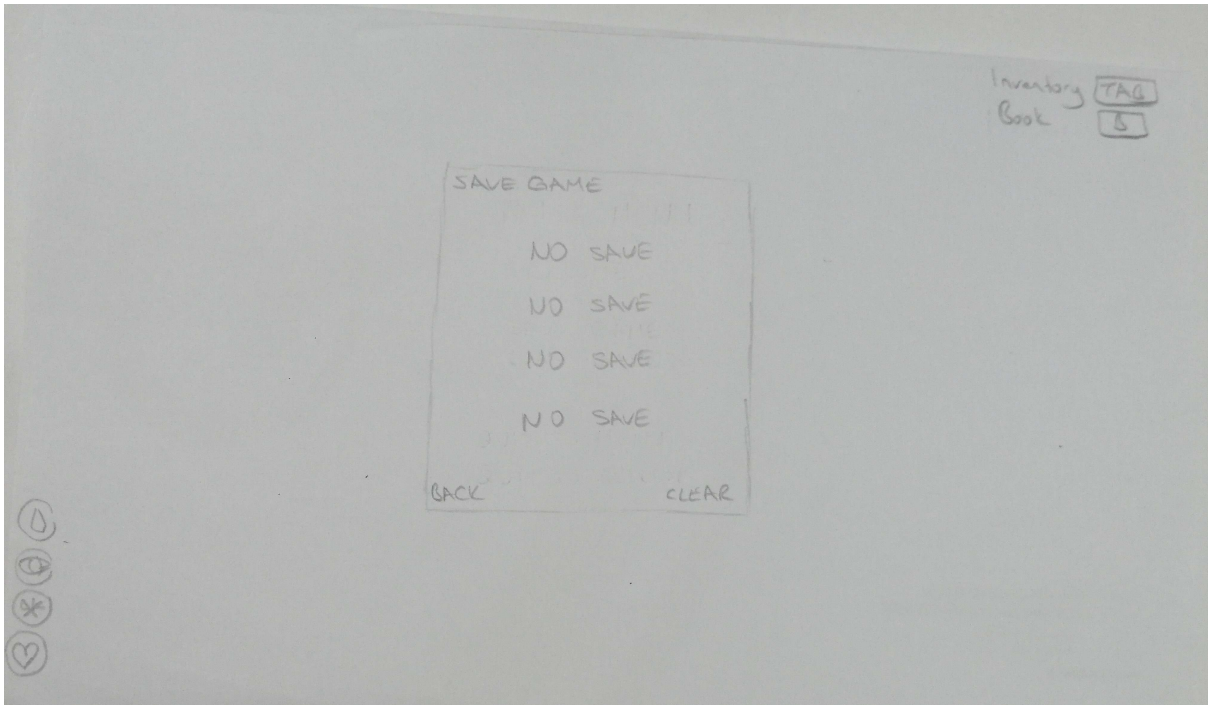


Figure 2: Save Game Menu

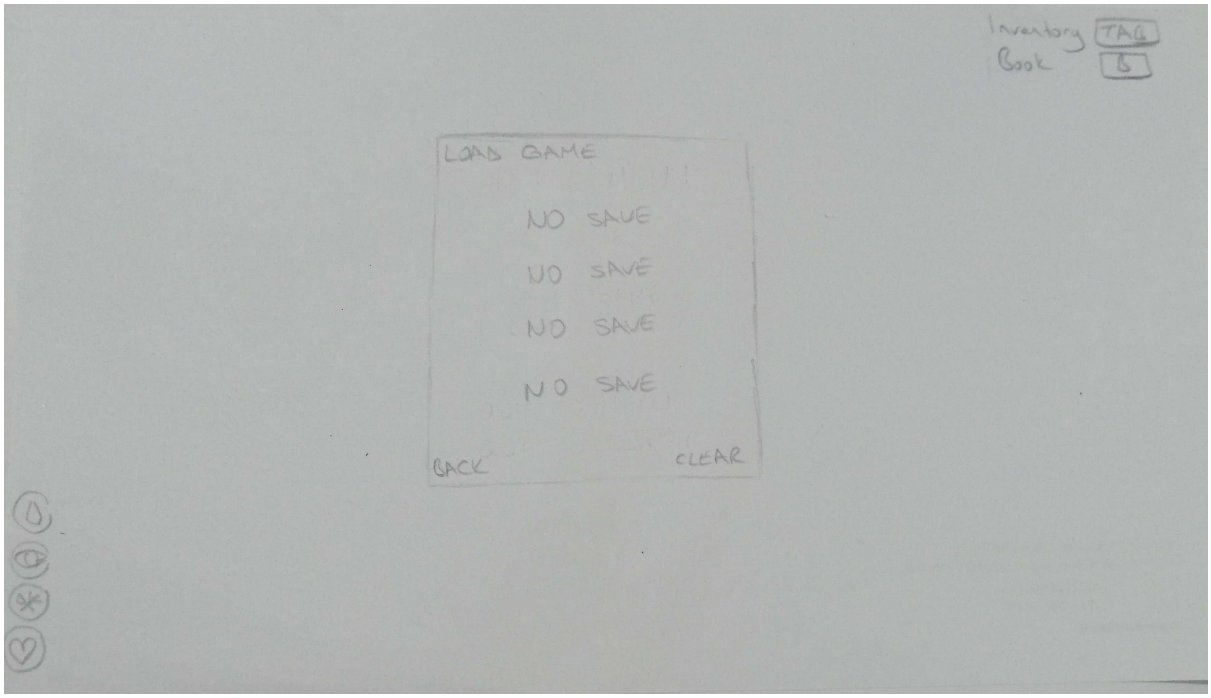


Figure 3: Load Game Menu

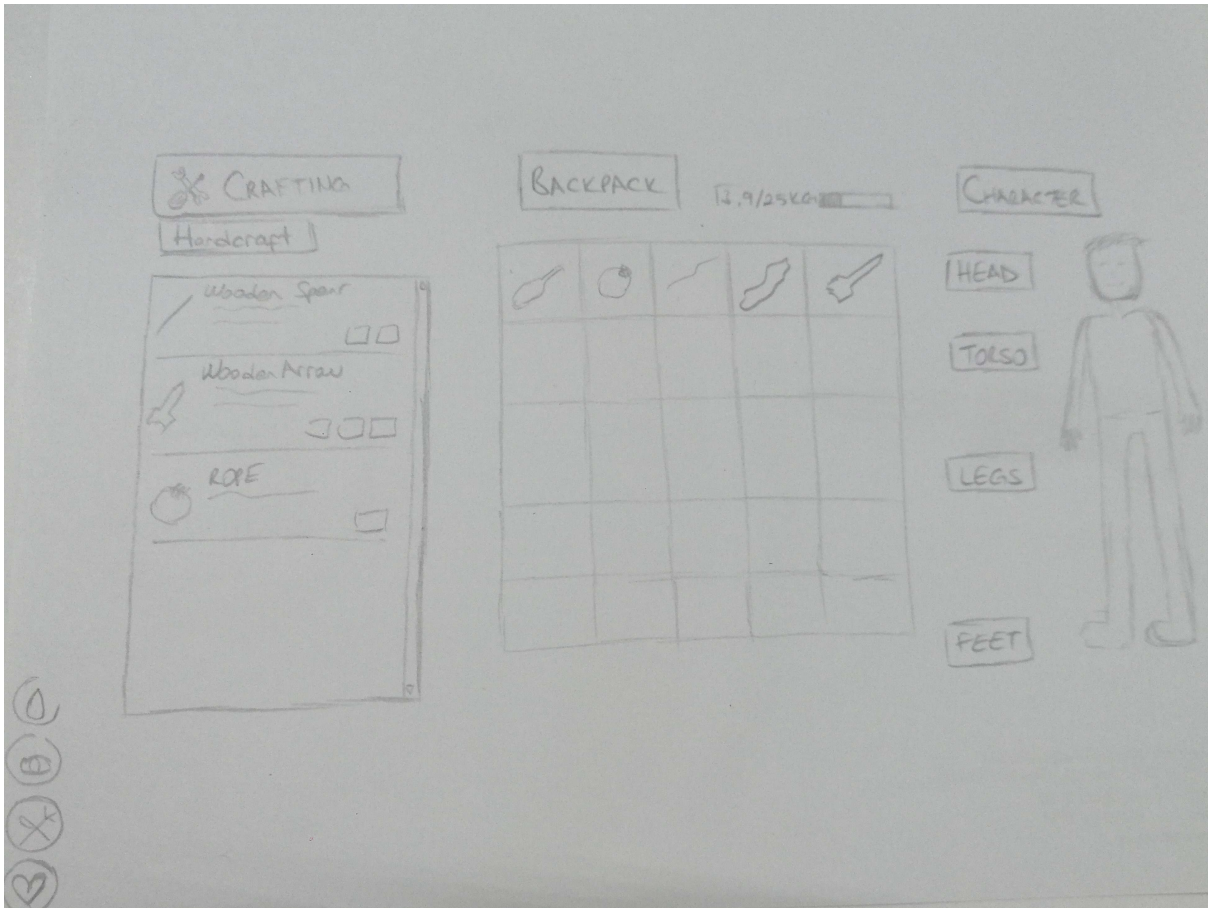


Figure 4: Inventory Menu

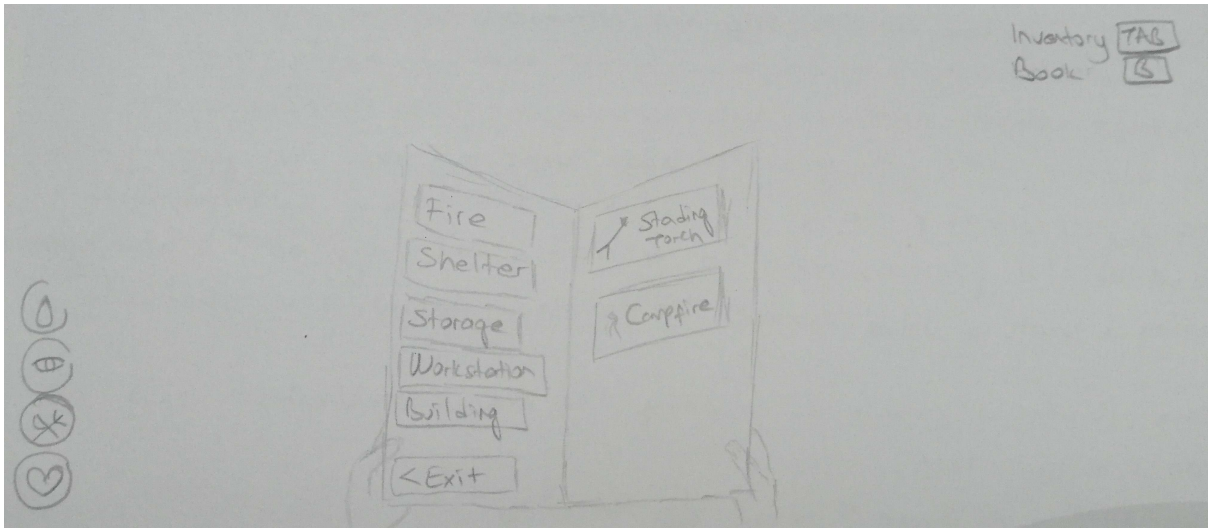


Figure 5: Building Menu

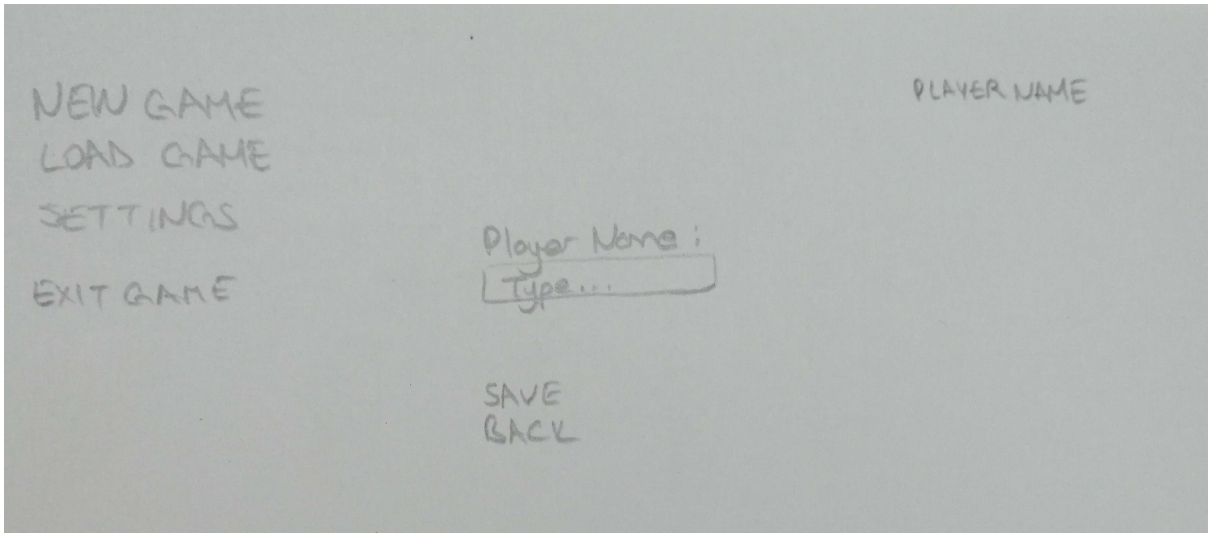


Figure 6: Main Menu

1.2 High Fidelity Mockups

These are the high-fidelity mockups that we made on the Canva website: [1]

Figure 7: Pause Menu

Figure 8: Save Game Menu

Figure 9: Load Game Menu

Figure 10: Inventory Menu

Figure 11: Building Menu

Figure 12: Main Menu

2 Explanation of the Implementation and the Methods

2.1 Implementation

We choose Unity engine for implementation because it is a popular engine for game development due to its many benefits and features. First and foremost, Unity is a cross-platform game engine, meaning that games developed with Unity can be deployed on multiple platforms including PC, mobile, consoles, and web browsers. This allows game developers to reach a wider audience and potentially increase the popularity and profitability of their games.

In addition to its cross-platform capabilities, Unity is also known for being easy to use, even for developers who are new to game development. Its intuitive interface and extensive documentation make it easy for developers to quickly get up to speed with the engine and start creating games. Furthermore, Unity has a large asset library, containing a wide range of pre-made game assets such as 3D models, textures, sound effects, and music. This can save developers a lot of time and effort, as they don't need to create everything from scratch.

Unity also has a strong community of developers who share tips, advice, and resources with each other. This community includes both professional game developers and hobbyists, making it a great resource for developers of all skill levels. Developers can ask questions, share their work, and collaborate with others to create better games.

Finally, Unity has a number of powerful features that make it a flexible choice for creating a wide range of games and interactive content. These features include support for 2D and 3D graphics, physics simulation, artificial intelligence, networking, and virtual and augmented reality. With these features, developers can create games and interactive experiences that are engaging and immersive.

2.2 Methods

Unity is a popular game engine that provides a wide range of tools for creating games, including a modular system for organizing and managing game objects and their behaviors. Here's an explanation of Unity modules, handlers, and behaviors:

2.2.1 Modules

Unity is built on a modular system, which means that it is made up of a series of interconnected modules that each serve a specific purpose. These modules include graphics, physics, input, audio, scripting, and more. Each module can be customized and extended to meet the specific needs of your game.

In our game, there are eight modules:
1 - Inventory Module: An Inventory Module in a game is responsible for managing the items that a player character can carry and use in the game.
The Inventory Module includes the following features:
- Item Collection: The module allows the player to collect items throughout the game world, such as weapons, ammunition, potions, keys, and other useful objects. The player can add these items to their inventory by picking them up or buying them from vendors.
- Inventory UI: The module displays the player's inventory on the game screen in an easy-to-understand format, typically as a list or grid of icons or images. The inventory UI includes features such as item descriptions, sorting options, and search functionality.

- Item Use: The module allows the player to use items in their inventory, such as equipping weapons, using potions, or unlocking doors. The player can typically access their inventory at any time during gameplay, but some games limit inventory use during combat or other critical situations.
- Inventory Management: The module allows the player to manage their inventory by dropping items they no longer need or selling them to vendors. The module also includes features such as item stacking, which allows the player to combine multiple identical items into a single inventory slot to save space.

2 - Health Module: Health Manager Module is responsible for managing the player character's health level and providing feedback to the player about their current health status.
The Health Manager Module includes the following features:

- Health Level: The module tracks the player character's current health level and updates it over time as the player takes damage. The health level is usually displayed on the player's screen as a visual indicator or as a numerical value.
- Damage and Healing: The module tracks the damage taken by the player character from enemy attacks, falls, or other hazards. The module also tracks the effects of healing items or abilities that restore the player's health.
- Death and Respawn: The module triggers the player's death if their health level drops to zero, and handles the process of respawning the player character at a safe location. The module also includes features such as respawn timers or penalties for dying.
- Health Effects: The Health Manager also applies effects to the player's character based on their current health level. For example, if the health level is very low, the player character's movement speed or accuracy is reduced.

3 - Hunger Module: Hunger Manager module in a game is responsible for managing the player character's hunger level and providing feedback to the player about their current hunger status.
The Hunger Manager module includes the following features:

- Hunger Level: The module tracks the player character's current hunger level and updates it over time as the player goes without food. The hunger level is displayed on the player's screen as a visual indicator.
- Hunger Reduction: The module reduces the player character's hunger level at a predetermined rate. The rate at which the hunger level decreases can also be influenced by the player's actions, such as running or using certain abilities.
- Food Consumption: The Hunger Manager allows the player to consume food items to restore their hunger level. The player can find or purchase food items throughout the game world, and consuming these items restores the hunger level based on the food's nutritional value.
- Hunger Effects: The Hunger Manager can also apply effects to the player character based on their current hunger level. For example, if the hunger level is very low, the player character's movement speed or attack power is reduced.

4 - Audio Module: Audio Module is responsible for managing the sound effects and music that

play during gameplay.
The Audio Module includes the following features:
- Sound Effects: The module plays a variety of sound effects during gameplay, such as footsteps, weapon sounds, explosion sounds, and environmental sounds. These sounds are triggered by player actions or events in the game world.
- Music: The module plays background music during gameplay, such as ambient music or dynamic music that changes based on the player's actions or the current situation in the game. The music can help to set the tone and atmosphere of the game, and can also provide feedback to the player about their progress and achievements.
- Audio Settings: The module allows the player to adjust the volume and other audio settings to their preferences. These settings include options such as volume, sound quality, surround sound, and language settings.
- Audio Management: The module manages the resources and memory used for audio playback, ensuring that sounds and music are played smoothly and without interruption. This includes features such as audio compression, resource loading, and audio streaming.

5 - Camera Module: Camera Module is responsible for managing the player's point of view and displaying the game world on the player's screen.
The Camera Module includes the following features:
- Camera Movement: The module controls the movement of the camera in the game world, allowing the player to look around and explore their surroundings. The camera movement is controlled by the player directly or by the game's mechanics such as cutscenes or scripted events.
- Camera Settings: The module allows the player to adjust camera settings such as field of view, camera sensitivity, and camera distance. These settings help the player to customize their viewing experience and optimize the gameplay.
- Camera Effects: The module applies visual effects to the camera, such as blur, depth of field, or motion blur. These effects enhance the immersion and realism of the game world and add visual interest.

6 - Sleep Module: Sleep Module is responsible for managing the player character's sleep needs and providing feedback to the player about their current sleep status.
The Sleep Module includes the following features:
- Sleep Needs: The module tracks the player character's current sleep needs and updates it over time. The sleep needs are displayed on the player's screen as a visual indicator.
- Sleep Deprivation: The module tracks the effects of sleep deprivation on the player's character, such as reduced performance, reduced accuracy, or slower movement. The effects of sleep deprivation are increased by staying awake for longer periods of time, or by engaging in activities that require high levels of concentration or physical exertion.
- Sleeping: The module allows the player character to sleep in a bed or other designated location to restore their sleep needs. The module also tracks the time spent sleeping and provides

feedback to the player about the quality of their sleep.
- Sleep Management: The module manages the resources and memory used for sleep management, ensuring that sleep needs and effects are updated smoothly and without interruption.

7 - Movement Module: Movement Module is responsible for managing the player character's movement and providing feedback to the player about their current movement status.
The Movement Module includes the following features:
- Movement Controls: The module manages the player character's movement controls, including walking, running, jumping, crouching, and other movements.
- Movement Physics: The module applies physics to the player character's movements, such as gravity, friction, and momentum. This physics affects the player character's speed, acceleration, and ability to change direction.
- Movement Animation: The module manages the player character's movement animation, providing visual feedback to the player about their current movement status. This animation adds realism and immersion to the game world.

8 - Building Module: Building Module is responsible for allowing the player to create, customize, and manage structures within the game world.
The Building Module includes the following features:
- Building Controls: The module manages the controls used to build structures within the game world, including placing and removing blocks, rotating objects, and adjusting placement settings.
- Building Materials: The module manages the resources and materials used to build structures within the game world, such as wood, stone, or metal. The player needs to gather or purchase these materials in order to build structures.
- Building Customization: The module allows the player to customize the appearance and functionality of their structures, such as by adding doors, windows, or other features. The player may be able to choose from a variety of building options and styles.
- Building Management: The module manages the resources and memory used for building management, ensuring that structures are created and updated smoothly and without interruption.

2.2.2 Handlers

In Unity, a handler is a type of script that controls the behavior of an object in your game. Handlers can be used to respond to events, such as mouse clicks or keyboard inputs, and they can be used to modify the properties of game objects, such as their position, rotation, and scale. Handlers are often used in conjunction with Unity's event system, which allows you to trigger events in response to user actions or other events in your game.

There are some handlers from our project :

- Player Look Handler: Handles look updating the look direction of a character.

- Item Drop Handler: Handles item pickup and dropping.

- Wieldable Survival Book Handler: Handles enabling and disabling the survival book wieldable.

- Crafting Handler: Handles item crafting and sound that will be played after crafting an item.

- Fall Damage Handler: Handles dealing fall damage to the character based on the impact velocity.

2.2.3 Behaviors

Behaviors in Unity are components that define the functionality of a game object. For example, you might have a behavior that controls the movement of a player character, or a behavior that defines the behavior of an enemy AI. Behaviors can be attached to game objects to add functionality to them, and they can be customized to meet the specific needs of your game.

In summary, Unity modules provide the building blocks for creating games, handlers control the behavior of objects in your game, and behaviors define the functionality of game objects. Understanding these concepts is essential for creating complex and engaging games in Unity.

There are some handlers from our project :

- Player Movement Input Behavior: Delegates movement input (using the new Unity Input System) to the ICharacterMover module.

- Player Building Input Behavior: Delegates building input (using the new Unity Input System) to the IBuildController module.

- Character Build Effects Behavior: Plays effects based on the IBuildController events.

- Character Sleep Effects Behavior: Plays effects based on the ISleepHandler events.

- Human Sounds Manager Behavior: Responsible for listening to events raised by other modules (e.g. health, movement, etc.) and playing "humanoid" sounds accordingly.

- Footsteps Manager Behavior: Responsible for playing footsteps sound based on the current IMover's module step length.

3 Functionality of the Written Codes

```
public interface ICharacter
{
    6 başvuru
    bool IsInitialized { get; }

    15 başvuru
    Transform ViewTransform { get; }
    9 başvuru
    Collider[] Colliders { get; }

    12 başvuru
    IAudioPlayer AudioPlayer { get; }
    26 başvuru
    IHealthManager HealthManager { get; }
    30 başvuru
    IInventory Inventory { get; }

    event UnityAction Initialized;


    31 başvuru
    bool TryGetModule<T>(out T module) where T : class, ICharacterModule;
    15 başvuru
    void GetModule<T>(out T module) where T : class, ICharacterModule;
    33 başvuru
    T GetModule<T>() where T : class, ICharacterModule;

    2 başvuru
    bool HasCollider(Collider collider);

    #region Monobehaviour
    0 başvuru
    GameObject gameObject { get; }
    12 başvuru
    Transform transform { get; }
    #endregion
}
```

Figure 13: ICharacter Class Code

This code defines an interface named ICharacter that specifies the properties and methods that a character in the game should have. The interface includes properties for determining whether the character has been initialized, the character's view transform, and an array of colliders that belong to the character. It also includes properties for an audio player, health manager, and inventory. The interface defines methods for obtaining modules of a certain type, checking whether a collider belongs to the character, and getting the game object and transforming of the character.

```
Unity Betiği | 3 başvuru
public class Character : MonoBehaviour, ICharacter
{
    3 başvuru
    public bool IsInitialized { get; private set; }

    // The property returns a reference to the Transform component named m_View.
    1 başvuru
    public Transform ViewTransform => m_View;
    4 başvuru
    public Collider[] Colliders { get; private set; }

    public event UnityAction Initialized;

    //[SerializeField, NotNull]
    private Transform m_View;

    1 başvuru
    public bool HasCollider(Collider collider)
    {
        for (int i = 0; i < Colliders.Length; i++)
        {
            if (Colliders[i] == collider)
                return true;
        }

        return false;
    }

    Unity İletisi | 2 başvuru
    protected virtual void Awake()
    {
        SetupBaseReferences();
    }

    Unity İletisi | 2 başvuru
    protected virtual void Start() // virtual alt sınıfların methodu yeniden tanımlayabilmesi için
    {
        IsInitialized = true;
        Initialized?.Invoke();
    }

    1 başvuru
    protected virtual void SetupBaseReferences()
    {
        Colliders = GetComponentsInChildren<Collider>(true);
    }
}
```

Figure 14: Character Class Code

This class is used to define the basic functionality of a character in the game and provides an implementation for the properties and methods defined in the ICharacter interface.

```
0 başvuru
public interface ILookHandler : ICharacterModule
{
    0 başvuru
    Vector2 ViewAngle { get; }
    0 başvuru
    Vector2 LookInput { get; }
    0 başvuru
    Vector2 LookDelta { get; }

    event UnityAction PostViewUpdate;

    0 başvuru
    void SetAdditiveLook(Vector2 look);
    0 başvuru
    void MergeAdditiveLook();


    // A method that will be used when the look handler needs input.
    0 başvuru
    void SetLookInput(LookHandlerInputDelegate input);
}


// A delegate that will be used when the look handler needs input.
public delegate Vector2 LookHandlerInputDelegate();
```

Figure 15: ILookHandler Class Code

This interface is used to define the properties and methods that a look handler should have in a character system for controlling the character's view or camera in a game. However, this interface does not provide any implementation for the properties and methods defined in the interface

```
using IdenticalStudios.InputSystem.Behaviours;
using IdenticalStudios;
using UnityEngine;

namespace IdenticalStudios.InputSystem.Behaviours
{
    Unity Betiği | 0 başvuru
    public abstract class CharacterInputBehaviour : InputBehaviour
    {
        protected Character Character;

        private bool m_Initialized;

        2 başvuru
        protected virtual void OnBehaviourEnabled(ICharacter character) { }
        1 başvuru
        protected virtual void OnBehaviourDisabled(ICharacter character) { }

        Unity İletisi | 3 başvuru
        protected sealed override void OnEnable()
        {
            if (Character == null)
                Character = transform.root.GetComponentInChildren<Character>();

            if (Character == null)
            {
                Debug.LogError($"({gameObject.name}) No parent character found.");
                return;
            }

            if (Character.IsInitialized)
            {
                OnBehaviourEnabled(Character);
                m_Initialized = true;
                base.OnEnable();
            }
            else
                Character.Initialized += OnInitialized;
        }

        Unity İletisi | 2 başvuru
        protected sealed override void OnDisable()
        {
            if (!m_Initialized || UnityUtils.IsQuittingPlayMode)
                return;

            OnBehaviourDisabled(Character);

            base.OnDisable();
        }

        2 başvuru
        private void OnInitialized()
        {
            Character.Initialized -= OnInitialized;
            OnBehaviourEnabled(Character);
            m_Initialized = true;
            base.OnEnable();
        }
    }
}
```

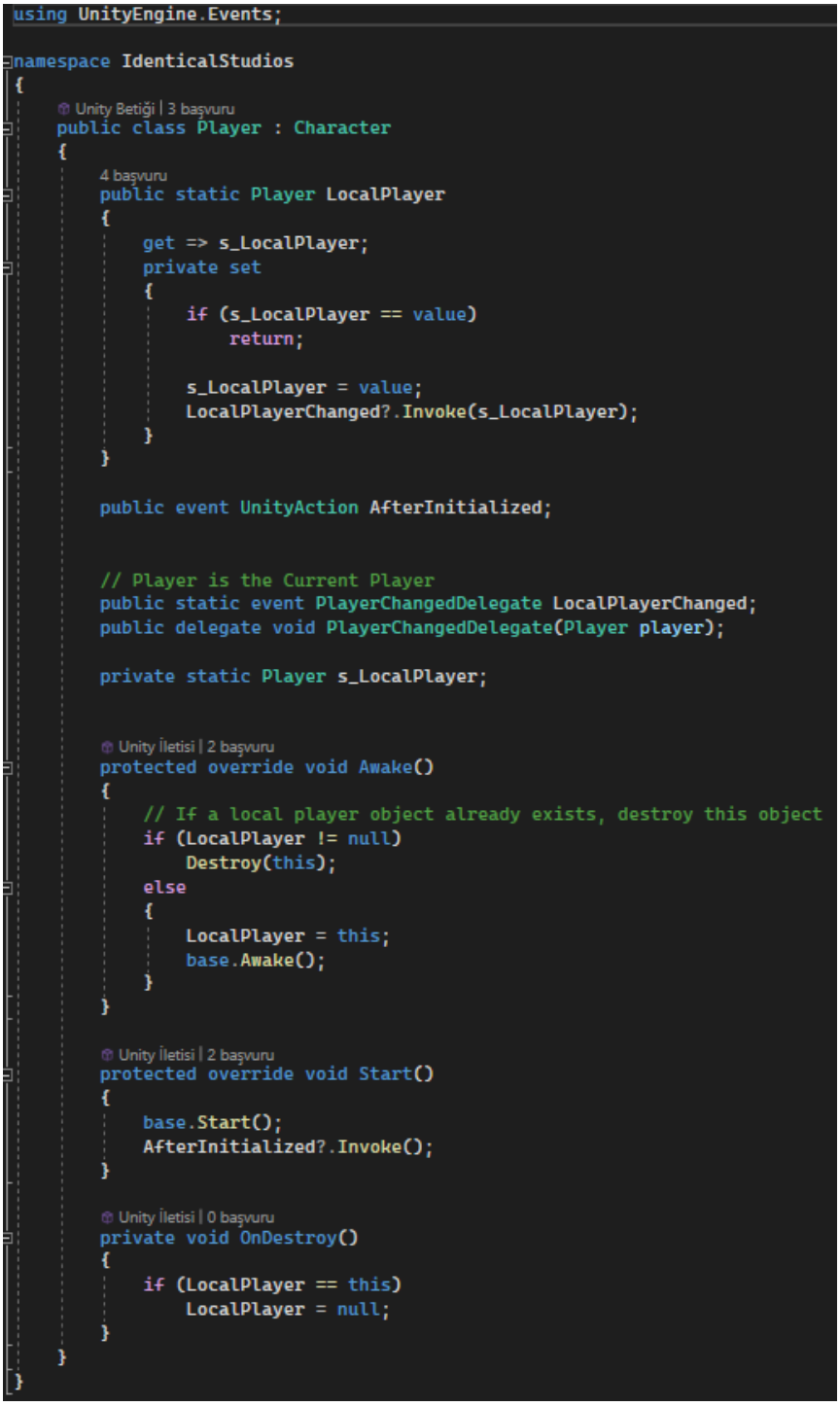
Figure 16: CharacterInputBehaviour Class Code

This code provides a base class for input behaviors that are specific to characters in the game. It ensures that the input behavior is only enabled if the associated character is initialized and provides hooks for derived classes to perform actions when the input behavior is enabled or disabled.

```
public abstract class InputBehaviour : MonoBehaviour
{

    private InputContext m_Context;

    4 başvuru
    protected virtual void OnInputEnabled() { }
    4 başvuru
    protected virtual void OnInputDisabled() { }
    1 başvuru
    protected virtual void TickInput() { }

    Unity İletisi | 3 başvuru
    protected virtual void OnEnable()
    {
        if (m_Context == null)
        {
            OnInputEnabled();
            return;
        }

        m_Context.ContextEnabled += OnInputEnabled;
        m_Context.ContextDisabled += OnInputDisabled;

        if (m_Context.IsEnabled)
            OnInputEnabled();
    }

    Unity İletisi | 2 başvuru
    protected virtual void OnDisable()
    {
        if (m_Context == null)
        {
            OnInputDisabled();
            return;
        }

        m_Context.ContextEnabled -= OnInputEnabled;
        m_Context.ContextDisabled -= OnInputDisabled;

        if (m_Context.IsEnabled)
            OnInputDisabled();
    }

    0 başvuru
    protected bool IsContextEnabled() => m_Context.IsEnabled;

    Unity İletisi | 0 başvuru
    private void Update()
    {
        if (m_Context == null || m_Context.IsEnabled)
            TickInput();
    }
}
```

Figure 17: InputBehaviour Class Code

This code is an abstract base class for input behavior components in Unity. It defines virtual methods for handling input enabling, disabling, and ticking (updating) input. It also has a private field for an input context, which can be used to enable or disable input for a group of input behavior components at once. When an input behavior component is enabled or disabled, it subscribes or unsubscribes from the input context's ContextEnabled and ContextDisabled events, respectively, and calls the appropriate OnInputEnabled or OnInputDisabled virtual methods. The TickInput method is called once per frame while the input behavior component is enabled, allowing for per-frame input processing. The IsContextEnabled method can be used to check if the input context is currently enabled.



```
using UnityEngine.Events;

namespace IdenticalStudios
{
    Unity Betiği | 3 başvuru
    public class Player : Character
    {
        4 başvuru
        public static Player LocalPlayer
        {
            get => s_LocalPlayer;
            private set
            {
                if (s_LocalPlayer == value)
                    return;

                s_LocalPlayer = value;
                LocalPlayerChanged?.Invoke(s_LocalPlayer);
            }
        }

        public event UnityAction AfterInitialized;


        // Player is the Current Player
        public static event PlayerChangedDelegate LocalPlayerChanged;
        public delegate void PlayerChangedDelegate(Player player);

        private static Player s_LocalPlayer;


        Unity İletisi | 2 başvuru
        protected override void Awake()
        {
            // If a local player object already exists, destroy this object
            if (LocalPlayer != null)
                Destroy(this);
            else
            {
                LocalPlayer = this;
                base.Awake();
            }
        }

        Unity İletisi | 2 başvuru
        protected override void Start()
        {
            base.Start();
            AfterInitialized?.Invoke();
        }

        Unity İletisi | 0 başvuru
        private void OnDestroy()
        {
            if (LocalPlayer == this)
                LocalPlayer = null;
        }
    }
}
```

Figure 18: Player Class Code

This class represents the local player in a game and provides functionality for setting up and initializing the player. It is also used to manage the local player object in a game and provides an implementation for the properties and methods required to manage it.

```
public class LookInput : CharacterInputBehaviour
{
    [Title("Actions")]

    [SerializeField]
    private InputActionReference m_LookInput;

    private ILookHandler m_LookHandler;


    3 başvuru
    protected override void OnBehaviourEnabled(ICharacter character)
    {
        character.GetModule(out m_LookHandler);
    }

    5 başvuru
    protected override void OnInputEnabled()
    {
        m_LookInput.Enable();
        m_LookHandler.SetLookInput(GetInput);
    }

    5 başvuru
    protected override void OnInputDisabled()
    {
        m_LookInput.TryDisable();
        m_LookHandler.SetLookInput(null);
    }

    1 başvuru
    private Vector2 GetInput()
    {
        Vector2 lookInput = m_LookInput.action.ReadValue<Vector2>() * 0.1f;
        lookInput.ReverseVector();

        return lookInput;
    }
```

Figure 19: LookInput Class Code

This code defines a LookInput class that inherits from CharacterInputBehaviour. It provides functionality to handle look input for a character in a game. When the behavior is enabled, it retrieves the ILookHandler component from the character and enables the m_LookInput action.

```
public interface IHealthManager : ICharacterModule
{
    6 başvuru
    bool IsAlive { get; }

    13 başvuru
    float Health { get; }
    4 başvuru
    float PrevHealth { get; }
    5 başvuru
    float MaxHealth { get; set; }

    event UnityAction<float, DamageContext> DamageTakenFullContext;

    event UnityAction<float> DamageTaken;
    event UnityAction<float> HealthRestored;
    event UnityAction<float> MaxHealthChanged;

    event UnityAction Death;
    event UnityAction Respawn;


    9 başvuru
    void RestoreHealth(float healthRestore);
    7 başvuru
    void ReceiveDamage(float damage);
    4 başvuru
    void ReceiveDamage(float damage, DamageContext dmgContext);
}
```

Figure 20: IHealthManager Class Code

This code defines an interface called IHealthManager which extends the ICharacterModule interface. It declares a number of properties and events related to health management, including IsAlive, Health, PrevHealth, and MaxHealth. It also defines methods to restore health and receive damage, with the latter having an optional parameter of DamageContext to provide additional context about the damage received. The interface also includes events for damage taken, health restored, max health changed, death, and respawn.

4 Commenting, Indentation of the Codes

```
public interface ICharacter
{
    6 başvuru
    bool IsInitialized { get; }

    15 başvuru
    Transform ViewTransform { get; }

    // All colliders attached to the character
    9 başvuru
    Collider[] Colliders { get; }

    // The character's audio player module
    12 başvuru
    IAudioPlayer AudioPlayer { get; }

    // The character's health manager module
    26 başvuru
    IHealthManager HealthManager { get; }

    // The character's inventory module
    30 başvuru
    IInventory Inventory { get; }

    // Event triggered when the character has been initialized
    event UnityAction Initialized;

    // Attempts to get a module of the given type and returns whether it was successful
    31 başvuru
    bool TryGetModule<T>(out T module) where T : class, ICharacterModule;

    // Gets a module of the given type
    15 başvuru
    void GetModule<T>(out T module) where T : class, ICharacterModule;

    // Gets a module of the given type
    33 başvuru
    T GetModule<T>() where T : class, ICharacterModule;

    // Checks whether the character has the given collider attached
    2 başvuru
    bool HasCollider(Collider collider);

    0 başvuru
    GameObject gameObject { get; }
    12 başvuru
    Transform transform { get; }
}
```

Figure 21: ICharacter Class Code

```
Unity Betiği | 3 başvuru
public class Character : MonoBehaviour, ICharacter
{
    3 başvuru
    public bool IsInitialized { get; private set; }

    1 başvuru
    public Transform ViewTransform => m_View;

    4 başvuru
    public Collider[] Colliders { get; private set; }

    //event that is invoked when the character is initialized.
    public event UnityAction Initialized;

    private Transform m_View;

    //method that returns whether the character has a given collider.
    1 başvuru
    public bool HasCollider(Collider collider)
    {
        for (int i = 0; i < Colliders.Length; i++)
        {
            if (Colliders[i] == collider)
                return true;
        }

        return false;
    }

    //method called when the component is first added to the game object.
    Unity İletisi | 2 başvuru
    protected virtual void Awake()
    {

        SetupBaseReferences();
    }

    // A virtual method called after Awake and when the game object is enabled.
    Unity İletisi | 2 başvuru
    protected virtual void Start()
    {
        // Marks the character as initialized and invokes the Initialized event.
        IsInitialized = true;
        Initialized?.Invoke();
    }

    //method to set up base references for the character.
    1 başvuru
    protected virtual void SetupBaseReferences()
    {
        // Gets all the colliders attached to the character.
        Colliders = GetComponentsInChildren<Collider>(true);
    }
}
```

Figure 22: Character Class Code

```
using IdenticalStudios;
using UnityEngine;
using UnityEngine.Events;

namespace IdenticalStudios
{
    0 başvuru
    public interface ILookHandler : ICharacterModule
    {
        0 başvuru
        Vector2 ViewAngle { get; }
        0 başvuru
        Vector2 LookInput { get; }
        0 başvuru
        Vector2 LookDelta { get; }

        // This event is triggered after the character's view has been updated
        event UnityAction PostViewUpdate;

        // This method adds an additional look input to the character's view
        0 başvuru
        void SetAdditiveLook(Vector2 look);
        // This method merges all accumulated look input into the final view
        0 başvuru
        void MergeAdditiveLook();

        /// A method that will be used when the look handler needs input.
        0 başvuru
        void SetLookInput(LookHandlerInputDelegate input);
    }

    /// A delegate that will be used when the look handler needs input.
    public delegate Vector2 LookHandlerInputDelegate();
}
```

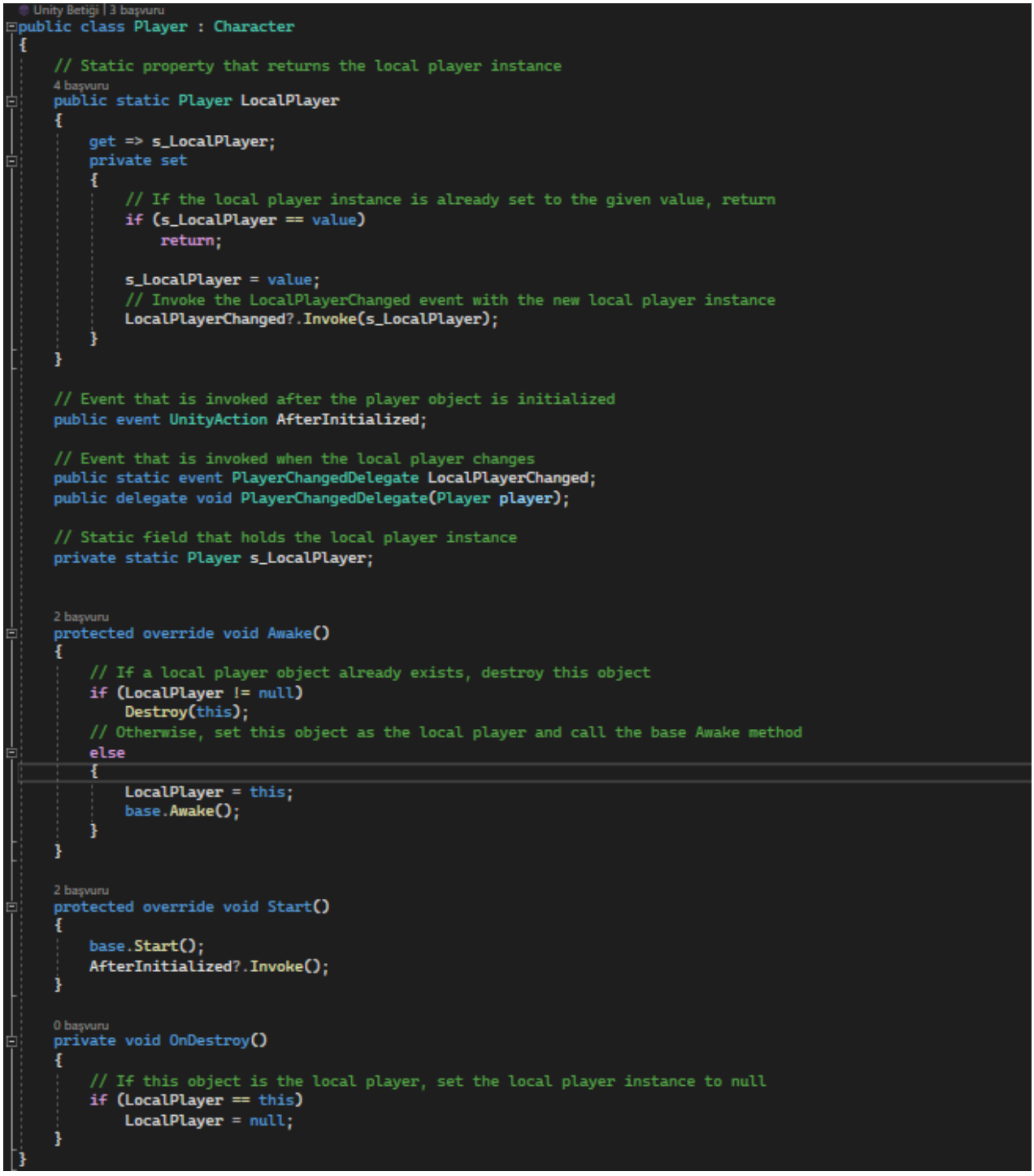
Figure 23: ILookHandler Class Code

```
namespace IdenticalStudios.InputSystem.Behaviours
{
    Unity Betiği | 0 başvuru
    public abstract class CharacterInputBehaviour : InputBehaviour
    {
        protected Character Character;
        private bool m_Initialized;

        // Called when this behaviour is enabled for the given character
        2 başvuru
        protected virtual void OnBehaviourEnabled(ICharacter character) { }
        // Called when this behaviour is disabled for the given character
        1 başvuru
        protected virtual void OnBehaviourDisabled(ICharacter character) { }

        // Called when this behaviour is enabled
        Unity İletisi | 3 başvuru
        protected sealed override void OnEnable()
        {
            if (Character == null)
                Character = transform.root.GetComponentInChildren<Character>();

            if (Character == null)
            {
                Debug.LogError($"({gameObject.name}) No parent character found.");
                return;
            }
            // If the character is already initialized, enable this behaviour
            if (Character.IsInitialized)
            {
                OnBehaviourEnabled(Character);
                m_Initialized = true;
                base.OnEnable();
            }
            // If the character is not yet initialized, wait for initialization
            else
                Character.Initialized += OnInitialized;
        }

        // Called when this behaviour is disabled
        // If this behaviour hasn't been initialized or the application is quitting, return
        Unity İletisi | 2 başvuru
        protected sealed override void OnDisable()
        {
            if (!m_Initialized || UnityUtils.IsQuittingPlayMode)
                return;

            OnBehaviourDisabled(Character);

            base.OnDisable();
        }

        // Called when the character is initialized
        2 başvuru
        private void OnInitialized()
        {
            Character.Initialized -= OnInitialized;
            OnBehaviourEnabled(Character);
            m_Initialized = true;
            base.OnEnable();
        }
    }
}
```

Figure 24: CharacterInputBehaviour Class Code

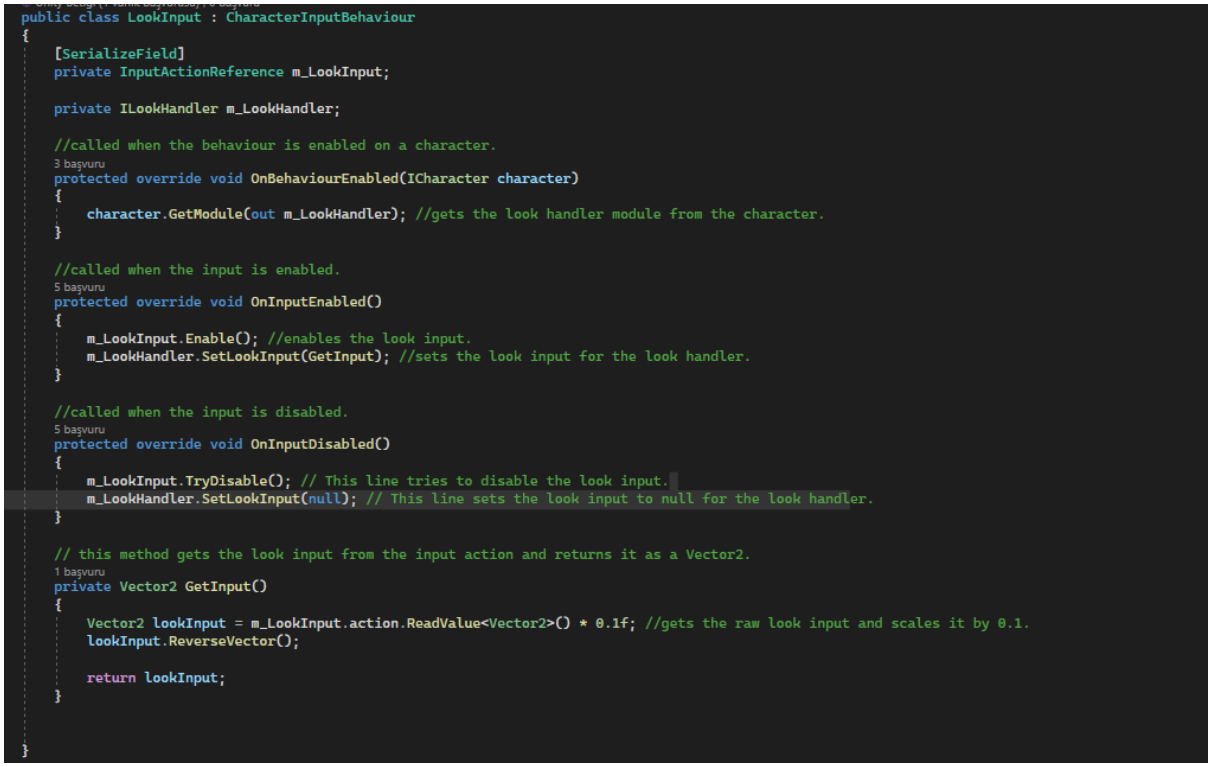
```
public abstract class InputBehaviour : MonoBehaviour
{
    private InputContext m_Context;

    /* OnInputEnabled() and OnInputDisabled() are called when the input is enabled or disabled,
     * respectively, while TickInput() is called every frame to process input */
    4 başvuru
    protected virtual void OnInputEnabled() { }
    4 başvuru
    protected virtual void OnInputDisabled() { }
    1 başvuru
    protected virtual void TickInput() { }

    Unity İletisi | 3 başvuru
    protected virtual void OnEnable()
    {
        /* This block of code checks if m_Context is null.If it is,
         * the OnInputEnabled() method is called and the function returns. */
        if (m_Context == null)
        {
            OnInputEnabled();
            return;
        }
        /* This block of code subscribes to the ContextEnabled and ContextDisabled events of
         * m_Context, and calls OnInputEnabled() if m_Context is currently enabled. */
        m_Context.ContextEnabled += OnInputEnabled;
        m_Context.ContextDisabled += OnInputDisabled;

        if (m_Context.IsEnabled)
            OnInputEnabled();
    }

    Unity İletisi | 2 başvuru
    protected virtual void OnDisable()
    {
        if (m_Context == null)
        {
            OnInputDisabled();
            return;
        }

        /* This block of code unsubscribes from the ContextEnabled and ContextDisabled events
         * of m_Context, and calls OnInputDisabled() if m_Context is currently enabled. */
        m_Context.ContextEnabled -= OnInputEnabled;
        m_Context.ContextDisabled -= OnInputDisabled;

        if (m_Context.IsEnabled)
            OnInputDisabled();
    }

    0 başvuru
    protected bool IsContextEnabled() => m_Context.IsEnabled;
    /* This method is called every frame and calls TickInput() if m_Context is either
     * null or enabled. This allows the inheriting class to process input every frame. */
    Unity İletisi | 0 başvuru
    private void Update()
    {
        if (m_Context == null || m_Context.IsEnabled)
            TickInput();
    }
}
```

Figure 25: InputBehaviour Class Code



```
Unity Betiği | 3 başvuru
public class Player : Character
{
    // Static property that returns the local player instance
    4 başvuru
    public static Player LocalPlayer
    {
        get => s_LocalPlayer;
        private set
        {
            // If the local player instance is already set to the given value, return
            if (s_LocalPlayer == value)
                return;

            s_LocalPlayer = value;
            // Invoke the LocalPlayerChanged event with the new local player instance
            LocalPlayerChanged?.Invoke(s_LocalPlayer);
        }
    }

    // Event that is invoked after the player object is initialized
    public event UnityAction AfterInitialized;

    // Event that is invoked when the local player changes
    public static event PlayerChangedDelegate LocalPlayerChanged;
    public delegate void PlayerChangedDelegate(Player player);

    // Static field that holds the local player instance
    private static Player s_LocalPlayer;


    2 başvuru
    protected override void Awake()
    {
        // If a local player object already exists, destroy this object
        if (LocalPlayer != null)
            Destroy(this);
        // Otherwise, set this object as the local player and call the base Awake method
        else
        {
            LocalPlayer = this;
            base.Awake();
        }
    }

    2 başvuru
    protected override void Start()
    {
        base.Start();
        AfterInitialized?.Invoke();
    }

    0 başvuru
    private void OnDestroy()
    {
        // If this object is the local player, set the local player instance to null
        if (LocalPlayer == this)
            LocalPlayer = null;
    }
}
```

Figure 26: Player Class Code



```
public class LookInput : CharacterInputBehaviour
{
    [SerializeField]
    private InputActionReference m_LookInput;

    private ILookHandler m_LookHandler;

    //called when the behaviour is enabled on a character.
    3 başvuru
    protected override void OnBehaviourEnabled(ICharacter character)
    {
        character.GetModule(out m_LookHandler); //gets the look handler module from the character.
    }

    //called when the input is enabled.
    5 başvuru
    protected override void OnInputEnabled()
    {
        m_LookInput.Enable(); //enables the look input.
        m_LookHandler.SetLookInput(GetInput); //sets the look input for the look handler.
    }

    //called when the input is disabled.
    5 başvuru
    protected override void OnInputDisabled()
    {
        m_LookInput.TryDisable(); // This line tries to disable the look input.
        m_LookHandler.SetLookInput(null); // This line sets the look input to null for the look handler.
    }

    // this method gets the look input from the input action and returns it as a Vector2.
    1 başvuru
    private Vector2 GetInput()
    {
        Vector2 lookInput = m_LookInput.action.ReadValue<Vector2>() * 0.1f; //gets the raw look input and scales it by 0.1.
        lookInput.ReverseVector();

        return lookInput;
    }


}
```

Figure 27: LookInput Class Code

```
public interface IHealthManager : ICharacterModule
{
    6 başvuru
    bool IsAlive { get; }
    13 başvuru
    float Health { get; }
    4 başvuru
    float PrevHealth { get; }
    5 başvuru
    float MaxHealth { get; set; }

    // This event is triggered when the character takes damage and provides the damage context
    event UnityAction<float, DamageContext> DamageTakenFullContext;

    // This event is triggered when the character takes damage and provides only the damage value
    event UnityAction<float> DamageTaken;

    // This event is triggered when the character's health is restored
    event UnityAction<float> HealthRestored;

    // This event is triggered when the character's maximum health is changed
    event UnityAction<float> MaxHealthChanged;

    // These events are triggered when the character dies and respawns
    event UnityAction Death;
    event UnityAction Respawn;

    // This method restores the health of the character by the specified amount
    9 başvuru
    void RestoreHealth(float healthRestore);
    // This method damages the character by the specified amount
    7 başvuru
    void ReceiveDamage(float damage);
    // This method damages the character by the specified amount and provides the damage context
    4 başvuru
    void ReceiveDamage(float damage, DamageContext dmgContext);
}
```

Figure 28: IHealthManager Class Code

5 Tests, Experimentation

5.1 Evaluation

For each user test, we stated the following:
Hello, we are Team Identical Studios and our game is a survival game. It is about surviving against hunger, water, nature, and animals. We hoped you could play it, try to survive, and enjoy it.
We created a demo scene and we put items in the scene for the task.
The task is the following:
- Run the game and start a new game.
- Open your inventory and wear the clothes.
- Find some sticks.
- Find some water bottles and drink them.
- Collect berries from bushes and eat them.
- Craft a spear from the handcrafting menu by using sticks.

- Place spear to holster and use it.
- Save your game and exit.

Evaluation 1 User1 is our classmate so he is an undergraduate student at Akdeniz University. He develops a lot of games. During the user test, we directed him through the protocol detailed above; there were a few usability problems User1 encountered:

Severity: Minor
User1 found the sound effects unnecessary.
User1 had difficulty reading the small text in the game, which caused eye strain and discomfort.

Severity: Major
User1 encountered a game-breaking glitch where the character became stuck in the environment and could not move. This required a restart of the game.

Evaluation 2 User2 is an experienced gamer. She plays a lot of games alone or with her friends. She plays at least three times a week. User2 is exactly the type of users IdenticalStudios is trying to target. During the user test, we directed her through the protocol detailed above; there were a few usability problems User2 encountered:

Severity: Minor
User2 was unable to locate any berries on the bushes, despite checking multiple bushes and waiting for a long time. This may indicate a bug or issue with the berry spawn rates.

Severity: Major
User2 experienced several crashes and freezes during the test, which interrupted gameplay and caused frustration.

Evaluation 3 User3 is a new player who is not very experienced with games. He has never played a survival game before. During the user test, we directed her through the protocol detailed above; there were a few usability problems User3 encountered:

Severity: Minor
User3 was confused by the controls for moving the character, as they were not explained clearly in the tutorial.
User3 had difficulty finding the crafting menu.

Severity: Major
User3 accidentally deleted the saved game data instead of loading it, resulting in lost progress.

5.2 Reflection

Based on the various evaluations of usability problems and their severities that we've examined, it is clear that there are many potential issues that may arise during user testing of a game or

other interactive product. Some of these problems may be minor and easily fixable, while others may be major and require significant effort to resolve.
As designers and developers, it is our responsibility to take these usability issues seriously and work to address them in a timely and effective manner. We must be attentive to user feedback and observations, and be willing to make changes and improvements as needed to ensure that our product is as user-friendly and enjoyable as possible.
It is important to remember that even minor usability problems can have a significant impact on a user's experience and that major issues can be extremely frustrating and off-putting. By taking a proactive approach to identifying and addressing these problems, we can create a more engaging, accessible, and enjoyable product for all users.

6 Guide & Documentation

Our game starts with a main menu. (Figure 30) It shows a text box for the player's name. User can write whatever it wants. After writing the name he/she can press the save button or cancel it by pressing the back button. If the user saves, the name will be shown in the right corner of the screen.(Figure 29)

Figure 29: Type Player Name

Figure 30: Main Menu

In the main menu, if the user presses the new game button, the game will be started. If the user presses the exit game button, the game will be closed. Also, there is a load game button for the continuation from the last saves. To use this there must be at least one save. (Figure 31)

Figure 31: Load Game Menu

In-game interface, there are some visual indicators in the left corner of the screen. Starting from the bottom, the first one is the health bar. It shows the player's health level. The next one is the hunger bar. It shows the player's hunger level. If the player eats food it will increase otherwise it will decrease with time. The next one is the energy bar. It shows the energy level. It will increase if the user sleeps over time. The last one is the thirst bar. It shows the thirst level and it will increase if the user drinks water otherwise it will decrease over time.

Also, there are some visual hints in the right-up corner about Inventory UI, holster UI, and building book. (Figure 32)

Figure 32: Game Interface

While playing, if the user presses the escape button it will open a pause menu. (Figure 33) There are buttons starting with resume. The resume button will close the pause menu and continue playing. The save game button opens another UI for the saving part.(Figure 34) The load game button opens another UI for the loading part. The quit to menu button will return to the main menu. The quit to the desktop button will close the game.

Figure 33: Pause Menu

The save game interface has four sections for saving. If you press one of these sections it will save the game. If there is a save it will override it. If you press the clear button from the

right-down corner it will clear all the saves.(Figure 34) If the user presses the back button it will be back to the pause menu. (Figure 33)

Figure 34: Save Game Menu

The load game interface has four sections for loading the game. (Figure 35) If you press one of these sections it will load the game. If there is unsaved progress it will be lost. If you press the clear button from the right-down corner it will clear all the loads. If the user presses the back button it will be back to the pause menu. (Figure 33)

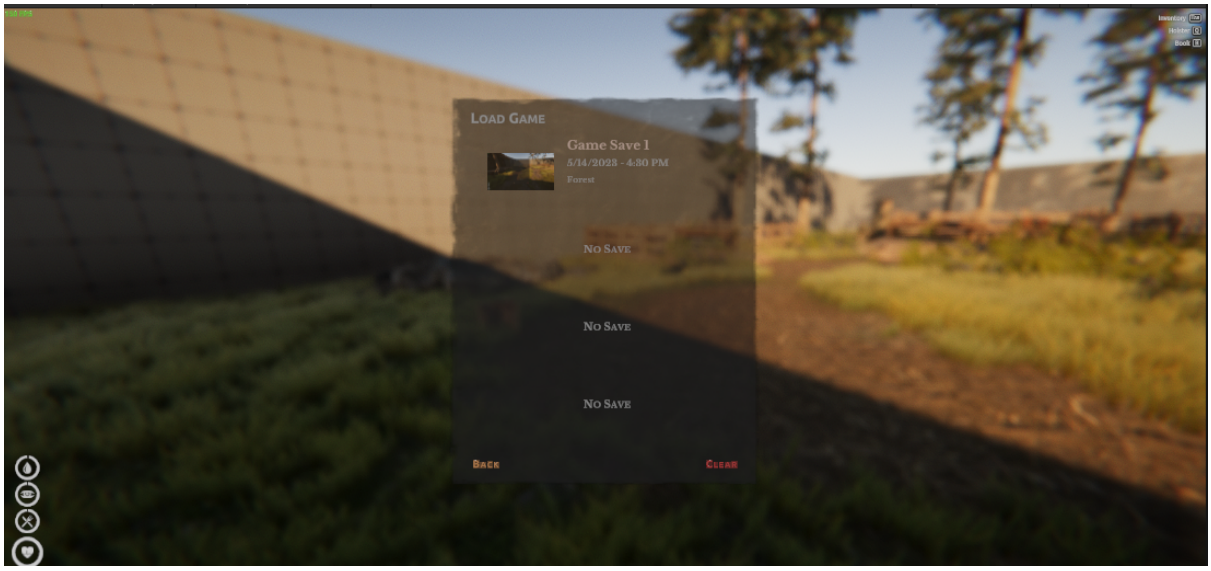


Figure 35: Load Game Menu

In the inventory interface, there is a backpack in the middle. There are item containers in the backpack. It shows the items that the user picked up. Also, there is a bar that shows the maximum carry weight with the current weight of the backpack.

At the left, there is Crafting UI and it shows the crafting list that can be made by hand. Also, it shows the recipes for craft items. If the user has enough ingredients he/she can craft items. At the right, there is Character UI and it shows the clothes on the character. There are sections for the head, torso, legs, and feet clothes. The user can remove them by dragging or can wear them by dragging. (Figure 36)

Figure 36: Inventory Menu

While playing pressing the 'b' button from the keyboard will open the building book. There are sections like fire, shelter, storage, workstation, and building. These sections have builds regarding their section. These builds need ingredients to build. If the user has enough ingredients, he/she can place them by clicking on them. Also, the book has an exit button to close the book.(Figure 37)

Figure 37: Book Menu

7 REFERENCES

References

[1] IdenticalStudios. *Canva*. 2023. URL: https://www.canva.com/design/DAFiuy07aWU/xhBRGMJWI_DeNMym0ybCKA/edit?utm_content=DAFiuy07aWU&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton.