Space Survival: Immersive VR Experience

Technical Implementation Report

Extended Reality Unit - ESIEE Paris

CHEN Michel
michel.chen@edu.esiee.fr

Academic Year 2024-2025
School: ESIEE Paris

Instructor: Badr TAJINI
badr.tajini@esiee.fr

June 8, 2025

Abstract

This comprehensive technical report documents the development of "Space Survival," an immersive virtual reality experience developed for the Extended Reality unit at ESIEE Paris. The project implements a single-player VR survival game where users must escape from a stranded spaceship through strategic resource management, environmental navigation, and problem-solving mechanics. Built using Unity 2022.3.21f1 with the Meta XR Simulator, the application demonstrates advanced VR interaction techniques, performance optimization strategies, and user comfort considerations. The implementation successfully maintains consistent performance above 30 FPS while providing an engaging narrative experience without inducing motion sickness. This report details the complete technical architecture, implementation challenges, optimization strategies, and educational outcomes achieved throughout the development process.

Contents

1 Executive Summary

1.1 Project Vision

Space Survival represents a comprehensive exploration of virtual reality development, combining technical excellence with creative storytelling. The project addresses the critical challenge of creating immersive VR experiences that maintain user comfort while delivering engaging gameplay mechanics. The core vision centers on demonstrating how sophisticated technical systems can work harmoniously to create seamless user experiences that push the boundaries of conventional VR interaction paradigms.

The project serves as both an educational exercise and a proof-of-concept for advanced VR development techniques. By focusing on the space survival theme, we establish a context that naturally requires complex interaction systems, environmental challenges, and resource management mechanics that showcase the full potential of virtual reality as a medium for interactive storytelling.

1.2 Key Achievements

- Successful implementation of a complete VR survival experience spanning multiple interaction domains
- Consistent frame rate above 30 FPS preventing cybersickness through systematic optimization
- Integration of advanced XR Interaction Toolkit features with custom extensions and modifications
- Creation of intuitive VR interaction systems that feel natural and responsive to user input
- Implementation of narrative-driven gameplay progression that seamlessly integrates story with mechanics
- Successful deployment using Meta XR Simulator enabling development without physical hardware dependencies

1.3 Technical Innovation

The project introduces several innovative technical solutions including custom XR socket interactions that provide enhanced control over object placement, advanced audio management systems that support both 2D UI feedback and spatial 3D audio, and optimized rendering pipelines specifically designed for VR environments that balance visual quality with performance requirements critical for user comfort.

2 Design Inspiration, User Journey, and Environment

2.1 Space Themed Assets and Inspiration

The visual and interactive elements of *Space Survival* are heavily influenced by classic and modern space survival games. My personal passion for the mysteries of space and the survival genre inspired both the aesthetics and core gameplay mechanics developed for this project.

2.1.1 Inspirational Source Games

Several notable games have served as direct or indirect inspiration for this work, notably for their immersive VR environments, survival mechanics, and resource management systems:

- **Star Shelter**

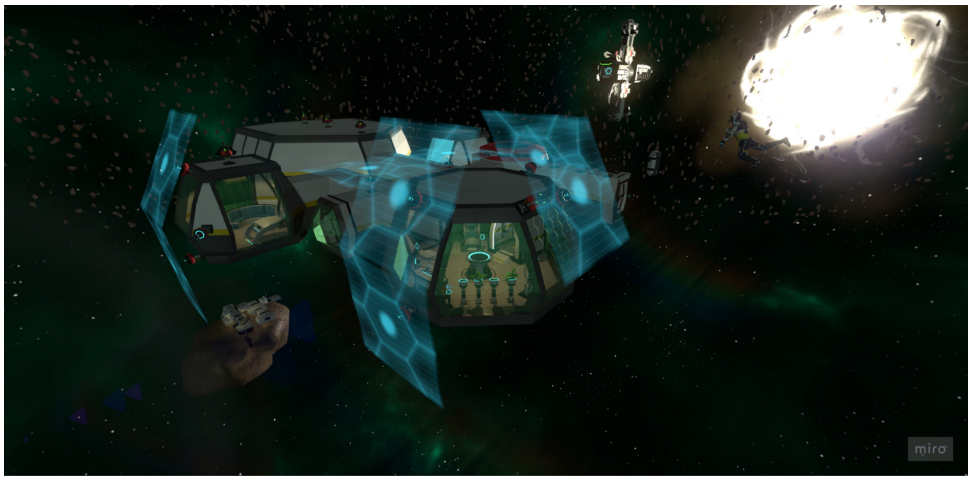

Figure 1: Star Shelter

`https://store.steampowered.com/app/698260/Star_Shelter/?l=english`
A virtual reality space survival game where players manage vital resources and explore abandoned stations to ensure their own survival.

- **Space Engineers**

Figure 2: Space Engineers

`https://store.steampowered.com/app/244850/Space_Engineers/`
A sandbox game set in space, focused on engineering, construction, exploration, and survival in an open world with realistic physics.

2.1.2 Assets and Functionalities

The design of *Space Survival* incorporates original and modified assets to evoke a convincing space setting, featuring:

- Space-themed props and environmental models (such as meteorites, debris, and modular ship interiors)
- Interactive spaceship controls and navigation systems inspired by realistic cockpits
- Survival UI elements: instructions and timer displays
- Atmospheric audio and visual effects, including dynamic lighting for enhanced immersion.

These elements collectively seek to immerse the player in a high-stakes survival scenario reflective of both my personal interests and the creative design lessons drawn from landmark titles in the space-survival game genre.

2.2 User Journey

This section outlines the typical experience of a player engaging with *Space Survival* from launching the application through to successfully completing (or failing) a play session. The user journey emphasizes the game's immersive flow, interaction design, and the challenges encountered.

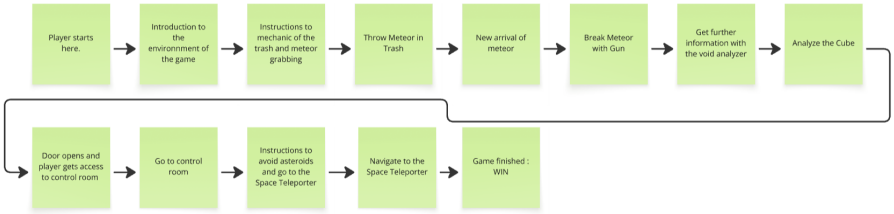

Figure 3: Sketch of the Outside Level of the Space Survival VR Game

2.3 Environment Design

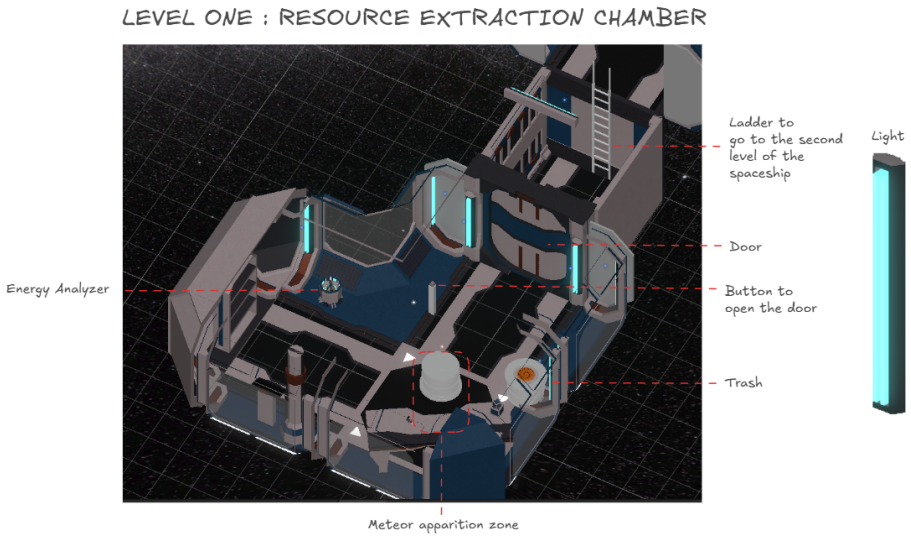

Figure 4: Sketch of the First Level of the Space Survival VR Game

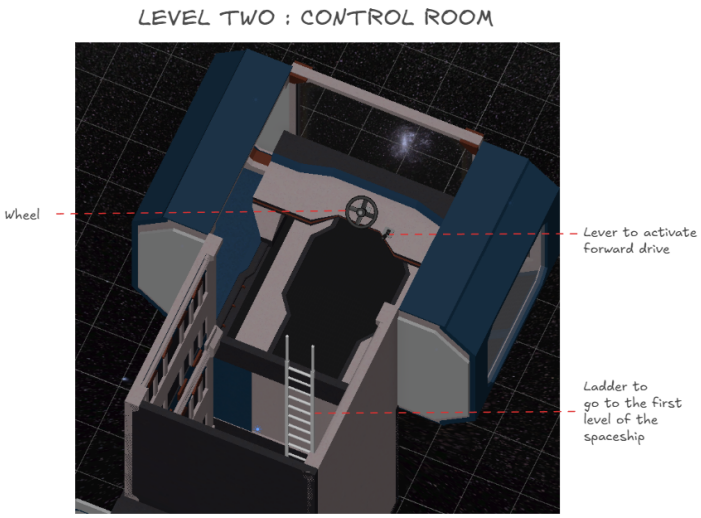

Figure 5: Sketch of the Second Level of the Space Survival VR Game

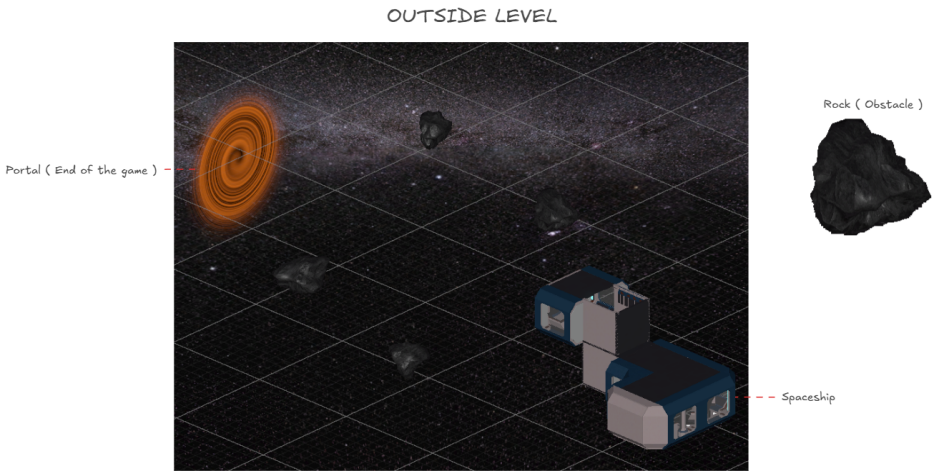

Figure 6: Sketch of the Outside Level of the Space Survival VR Game

3 Game Lore

Space Survival unfolds in a not-so-distant future, where humanity explores the cosmos in search of new worlds and vital resources. You assume the role of the engineer aboard the research ship **Artemis**, tasked with supporting a deep-space mining expedition on the edge of known space.

3.1 Disaster Strikes

During a routine survey, the *Artemis* is battered by a sudden meteor storm, breaching multiple hull sections and crippling life-support systems.

When you awake, the ship is eerily silent. Ressources are dwindling, critical systems flicker on the edge of failure. It's a race against time to survive and uncover what truly happened.

3.2 Environmental Storytelling

The *Space Survival* experience is designed to reveal its lore organically:

- **Audio logs**: Provide instructions, technical details, and personal reflections.

3.3 Your Mission

You must:

- Gather Energy cube.
- Restart the Spaceship by placing the Energy cube on the Energy Analyzer.
- Navigate the Spaceship to go to a portal without colliding with the Meteors.

The lore-driven approach gives every audio log, and clue a place in the larger narrative—heightening immersion and rewarding exploration as you fight for survival aboard the doomed Spaceship.

4 Implementation Details

4.1 Lighting, Rendering and Optimization

Achieving high visual fidelity while maintaining optimal performance is a major challenge in VR development. For **Space Survival**, a comprehensive lighting and rendering strategy was implemented, combining advanced baking, probe placement, object management, and physics optimizations.

4.1.1 Lighting Architecture and Shadow Occlusion

We applied a mixed approach using both baked and real-time lights to keep the scene dynamic where needed (e.g., the cockpit controls or flickering emergency lights) while using precomputed lighting and shadows for static areas. For better shadow realism and performance, an **occlusion effect** was applied to shadowed areas, blending baked ambient occlusion data into the lightmaps. This produces more convincing depth and contact points without taxing the GPU with real-time calculations.

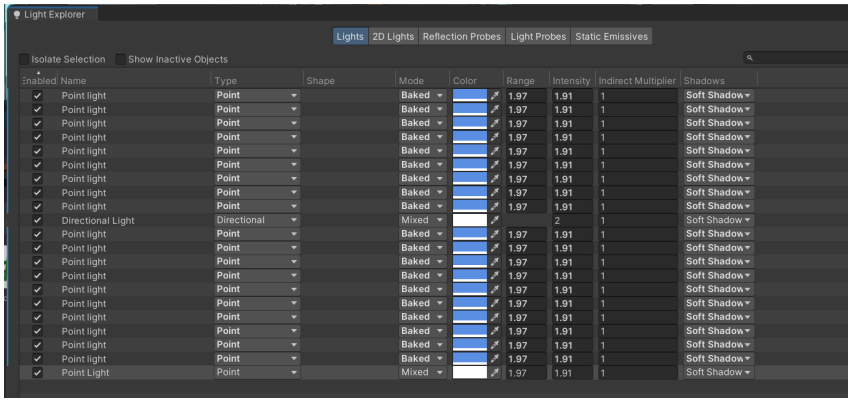

Figure 7: General lighting optimization workflow: static geometry, lightmaps, and probe placement.

4.1.2 Reflection and Light Probes

To improve the quality of indirect lighting and reflections for dynamic and static objects alike, we placed both **Reflection Probes** and **Light Probe Groups**.

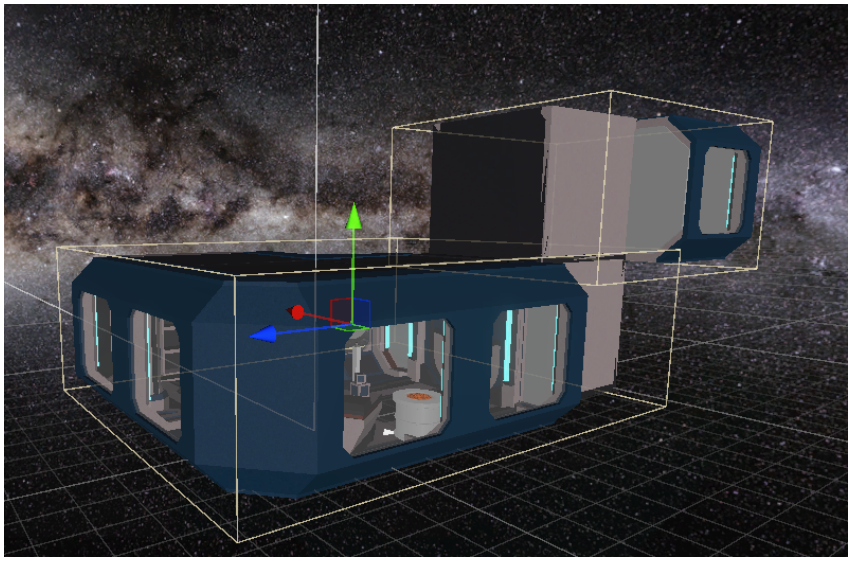

Figure 8: Reflection Probe: captures the environment's radiance for accurate real-time reflections on metal and glass surfaces.

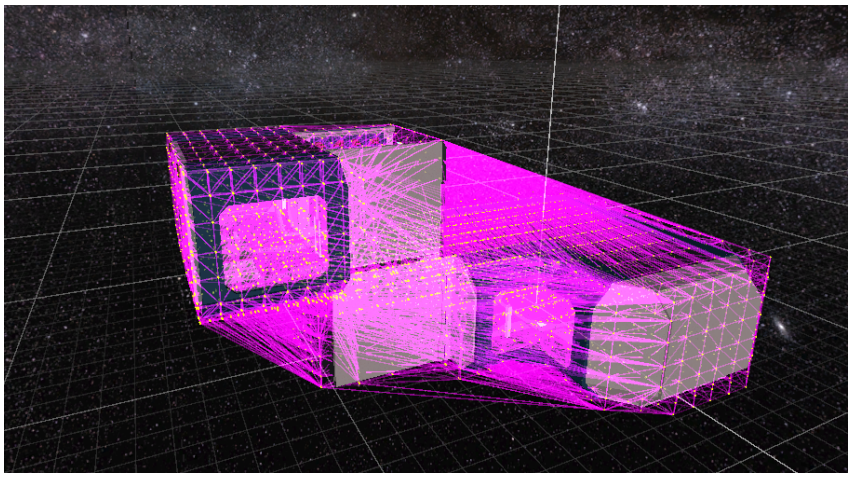

Figure 9: Light Probe Group: interpolates indirect lighting for dynamic/moving objects in the scene.

This allows moving/interactable elements to receive high-quality shading and reflections as they traverse the space, even though the main lighting is primarily baked.

4.1.3 Baked Lighting and Lightmaps

All static geometry (walls, floors, fixed props) use **baked lightmaps** with occlusion and shadowing computed in advance. This reduces real-time workload and increases overall frame rates. We previewed and tweaked the lightmap packing, texel density, and resolution to get crisp shadow boundaries without excess memory use.

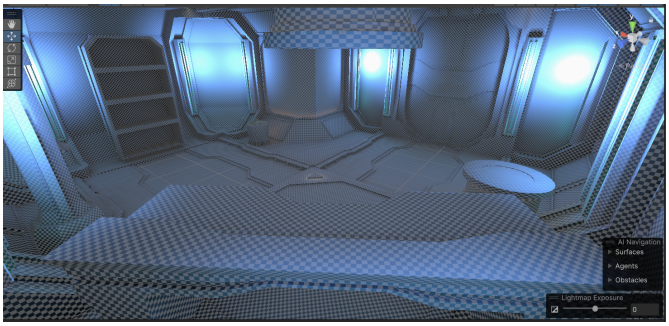

Figure 10: Baked Lightmap: visualization of texel layout and shadow information for static geometry.

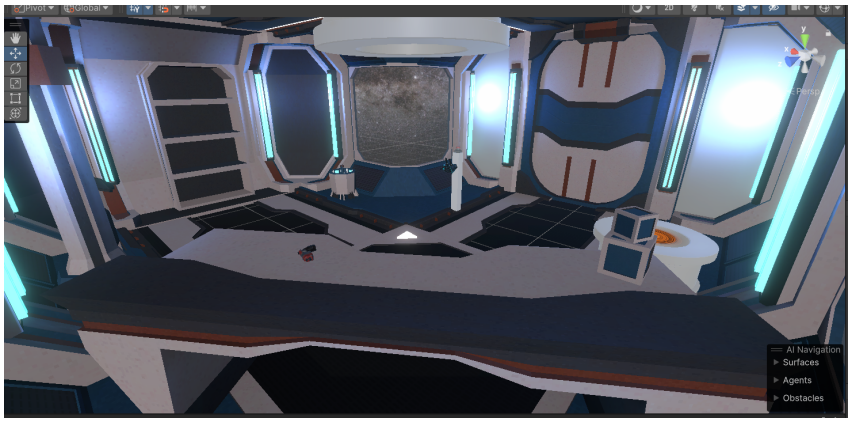

Figure 11: Baked Lighting Result: in-game result showing static shadowing and occlusion effects.

4.1.4 Static Batching and Object Separation

For maximum rendering and culling performance, all purely static objects were organized under a dedicated `Static` folder, and flagged as static in Unity. This enables Unity to aggressively batch draw calls, combine geometry, and pre-calculate visibility, dramatically reducing CPU+GPU load during play.

Dynamic and interactable objects (weapons, debris, controls) are kept separate to avoid lightmap or batching issues, and use probe-based lighting.

4.1.5 Physics and Miscellaneous Optimizations

All interactable objects had their physics mode set to `Discrete`, which avoids unnecessary collision evaluations while keeping interactions responsive—a crucial point in VR. Where appropriate, we reduced the number of physics layers and filtered unnecessary collisions between static/static or unimportant layers, further optimizing frame rate.

Summary of Rendering Optimizations:

- Mixed baked and dynamic lighting, with baked occlusion for realistic static shadowing
- Systematic use of Reflection and Light Probe Groups for high-quality dynamic lighting and reflections

- Separation of all static geometry—flagged as static for batching, culling, and pre-computed lighting
- Physics optimizations including `Discrete` collision modes and cleaned-up collision layers
- Result: steady frame rate above 30 FPS with visually rich, immersive environments suitable for VR

4.2 Audio Management System

The audio system serves as the cornerstone of immersive feedback throughout the Space Survival experience. This centralized architecture provides comprehensive sound management capabilities while maintaining optimal performance characteristics essential for VR applications.

4.2.1 AudioManager Core Architecture

The AudioManager implementation represents a sophisticated approach to game audio that balances functionality with performance. The system utilizes the Singleton design pattern to ensure global accessibility while preventing resource conflicts that could impact VR performance.

```
[System.Serializable]
public class Sound
{
    public string name;
    public AudioClip clip;
    [Range(0,1)]
    public float volume = 1;
    [Range(-3,3)]
    public float pitch = 1;
    public bool loop = false;
    public AudioSource source;

    public Sound()
    {
        volume = 1;
        pitch = 1;
        loop = false;
    }
}

public class AudioManager : MonoBehaviour
{
    public Sound[] sounds;
    public static AudioManager instance;

    void Awake()
    {
        // Singleton pattern implementation with persistence
        if (instance == null)
            instance = this;
        else
        {
```

```
            Destroy(gameObject);
            return;
        }

        DontDestroyOnLoad(gameObject);

        // Dynamic audio source creation and configuration
        foreach (Sound s in sounds)
        {
            if(!s.source)
                s.source = gameObject.AddComponent<AudioSource>();

            s.source.clip = s.clip;
            s.source.volume = s.volume;
            s.source.pitch = s.pitch;
            s.source.loop = s.loop;
        }
    }

    public void Play(string name)
    {
        Sound s = Array.Find(sounds, sound => sound.name == name);
        if (s == null)
        {
            Debug.LogWarning("Sound: " + name + " not found");
            return;
        }
        s.source.Play();
    }

    public void Stop(string name)
    {
        Sound s = Array.Find(sounds, sound => sound.name == name);
        s.source.Stop();
    }
}
```

Listing 1: Sound Data Structure and AudioManager Implementation

Detailed Analysis of Sound Class Structure:

The Sound class serves as a data container that encapsulates all properties necessary for comprehensive audio control. The `name` field provides string-based identification, enabling other game systems to trigger audio events using descriptive names rather than direct references. This approach significantly improves code maintainability and allows for dynamic audio content management.

The `volume` and `pitch` ranges are carefully selected to provide meaningful control without enabling destructive parameter values. The volume range of 0-1 maps directly to Unity's AudioSource volume parameter, while the pitch range of -3 to +3 allows for dramatic audio effects while maintaining audio quality.

The `loop` boolean provides essential control for ambient sounds and continuous effects, particularly important in the spaceship environment where engine sounds and environmental audio must persist throughout specific gameplay phases.

AudioManager Singleton Implementation Details:

The Singleton pattern implementation includes critical safety measures for VR development. The `DontDestroyOnLoad` call ensures audio continuity during scene transitions,

preventing jarring audio interruptions that could break immersion. The singleton enforcement prevents multiple AudioManager instances that could cause audio conflicts or performance degradation.

The dynamic AudioSource creation strategy optimizes memory usage by creating components only when needed, while the configuration loop ensures each Sound object has properly configured audio sources before gameplay begins. This approach prevents runtime errors and ensures consistent audio behavior across all game systems.

String-Based Audio Triggering:

The string-based audio identification system provides significant advantages for VR development. Other game systems can trigger audio without maintaining direct references to AudioClip objects, reducing coupling between systems and improving modularity. The warning system for missing sounds aids in debugging and content management during development.

4.2.2 UI Audio Integration System

The UI audio system provides consistent feedback for user interface interactions, supporting both traditional pointer-based input and VR controller input methods. This dual compatibility ensures the system functions correctly across different input paradigms.

```
public class UIAudio : MonoBehaviour, IPointerClickHandler,
    IPointerEnterHand ler, IPointerExitHandler
{
    [Header("Audio Configuration")]
    public string clickAudioName = "";
    public string hoverEnterAudioName = "";
    public string hoverExitAudioName = "";

    public void OnPointerClick(PointerEventData eventData)
    {
        if(clickAudioName != "" && AudioManager.instance != null)
        {
            AudioManager.instance.Play(clickAudioName);
        }
    }

    public void OnPointerEnter(PointerEventData eventData)
    {
        if (hoverEnterAudioName != "" && AudioManager.instance != null)
        {
            AudioManager.instance.Play(hoverEnterAudioName);
        }
    }

    public void OnPointerExit(PointerEventData eventData)
    {
        if (hoverExitAudioName != "" && AudioManager.instance != null)
        {
            AudioManager.instance.Play(hoverExitAudioName);
        }
    }
}
```

Listing 2: UI Audio Feedback Implementation

Interface-Based Event Handling:

The implementation of Unity's pointer event interfaces (`IPointerClickHandler`, `IPointerEnterHan` `IPointerExitHandler`) provides standardized interaction handling that works seamlessly with both mouse input and VR controller raycasting. This approach ensures consistent behavior across different input methods without requiring separate code paths.

Configurable Audio Feedback:

Each UI element can specify different audio clips for different interaction states, enabling rich audio feedback that enhances user experience. The string-based configuration allows designers to easily modify audio behavior without code changes, supporting rapid iteration during development.

Safety and Error Handling:

The null checks for AudioManager.instance and empty string validation prevent runtime errors that could disrupt VR experiences. These safety measures are particularly important in VR where any interruption to the audio system could break immersion or cause discomfort.

4.3 Meta XR Simulator Implementation and Package Setup

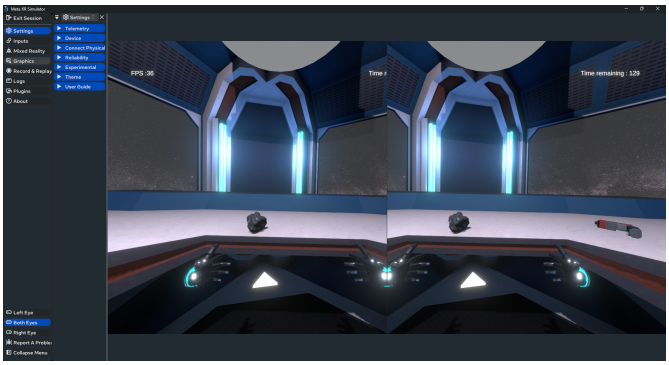

Figure 12: Meta XR Simulator: Both XR Vision Eyes Enabled

To enable VR development and testing without physical headset hardware, the **Meta XR Simulator** was implemented as the core runtime environment for prototyping, debugging, and evaluating gameplay in *Space Survival*. This toolset allowed full simulation of hand tracking, controller input, and VR headset movement within the Unity Editor.

4.3.1 Package Configuration

The following Unity packages were installed and configured, ensuring compatibility between the simulator, interaction toolkits, and Meta-specific SDKs:

Table 1: Unity XR and Meta XR Packages Used for Development

Package Name	Version
Meta - Voice SDK - Immersive Voice Commands	74.0.0
Meta MR Utility Kit	74.0.2
Meta XR All-in-One SDK	74.0.3
Meta XR Audio SDK	74.0.3
Meta XR Core SDK	76.0.0
Meta XR Haptics SDK	74.0.2
Meta XR Interaction SDK	76.0.0
Meta XR Interaction SDK Essentials	76.0.0
Meta XR Platform SDK	74.0.0
Meta XR Simulator	76.0.0
XR Core Utilities	2.3.0
XR Interaction Toolkit	2.5.4

4.3.2 Usage and Benefits

- **Meta XR Simulator** (76.0.0): Enabled in-editor VR simulation, providing rapid workflow for interaction testing and debugging without constant headset deployment.
- **XR Interaction Toolkit** (2.5.4) & **XR Core Utilities** (2.3.0): Provided cross-platform interaction, grabbable objects, socketing, ray interactor, and haptic feedback interfaces essential for the gameplay experience.
- **Meta XR SDK Suite** (Audio, Haptics, Platform, All-in-One, etc.): Supported hardware-specific features such as spatial voice, audio, haptics, and compatibility layers for Meta (Quest) headsets.

Implementation Notes:

- The simulator allowed for complete prototyping across all gameplay and UI flow (including hand model rendering, controller raycasting, button actuation, and comfort features) before deployment to actual Meta Quest hardware.
- All interactions developed with the XR Interaction Toolkit were compatible across simulator and real hardware, ensuring robust, hardware-agnostic code.
- Package versions were locked and tested for compatibility to avoid API changes or breaking updates during development.

This careful package curation and the use of the Meta XR Simulator was fundamental to efficient iteration, robust debugging.

4.4 XR Interaction Systems

The XR interaction systems form the foundation of player engagement within the Space Survival experience. These systems leverage Unity's XR Interaction Toolkit while providing custom extensions that address specific gameplay requirements.

4.4.1 Hand Presence Management System

Hand presence in VR environments requires careful management to maintain immersion while providing functional interaction feedback. The DisableGrabbingHandModel component addresses the common problem of hand model clipping through grabbed objects.

```
public class DisableGrabbingHandModel : MonoBehaviour
{
    [Header("Hand Model References")]
    public GameObject leftHandModel;
    public GameObject rightHandModel;

    [Header("Debug Information")]
    public bool debugMode = false;

    void Start()
    {
        // Ensure grab interactable component exists
        XRGrabInteractable grabInteractable = GetComponent<
    XRGrabInteractable>();
        if (grabInteractable == null)
        {
            Debug.LogError($"XRGrabInteractable component not found on
    {gameObject.name}");
            return;
        }

        // Register event listeners for grab state changes
        grabInteractable.selectEntered.AddListener(HideGrabbingHand);
        grabInteractable.selectExited.AddListener(ShowGrabbingHand);

        if (debugMode)
        {
            Debug.Log($"Hand visibility controller initialized for {
    gameObject.name}");
        }
    }

    public void HideGrabbingHand(SelectEnterEventArgs args)
    {
        // Identify which hand is grabbing based on controller tag
        string interactorTag = args.interactorObject.transform.tag;

        if (debugMode)
        {
            Debug.Log($"Hiding hand for interactor: {interactorTag}");
        }

        if (interactorTag == "Left Hand" && leftHandModel != null)
        {
            leftHandModel.SetActive(false);
```

```
        }
        else if (interactorTag == "Right Hand" && rightHandModel != null)
        {
            rightHandModel.SetActive(false);
        }
        else
        {
            Debug.LogWarning($"Unknown interactor tag or missing hand model: {interactorTag}");
        }
    }

    public void ShowGrabbingHand(SelectExitEventArgs args)
    {
        // Restore hand visibility when object is released
        string interactorTag = args.interactorObject.transform.tag;

        if (debugMode)
        {
            Debug.Log($"Showing hand for interactor: {interactorTag}");
        }

        if (interactorTag == "Left Hand" && leftHandModel != null)
        {
            leftHandModel.SetActive(true);
        }
        else if (interactorTag == "Right Hand" && rightHandModel != null)
        {
            rightHandModel.SetActive(true);
        }
    }
}
```

Listing 3: Advanced Hand Model Visibility Control

Event-Driven Hand Management:

This implementation demonstrates sophisticated understanding of VR interaction principles. The system responds to XR Interaction Toolkit events (`selectEntered` and `selectExited`) to create smooth transitions between visible and hidden hand states. This approach is essential for maintaining immersion because visible hand models clipping through grabbed objects creates visual artifacts that can break presence.

Tag-Based Hand Identification:

The tag-based identification system provides flexible hand detection that works with various XR controller setups. By using string tags rather than direct component references, the system remains modular and can adapt to different hand tracking implementations or controller configurations.

Error Handling and Debugging:

The comprehensive error handling prevents null reference exceptions that could disrupt VR experiences. The debug mode functionality aids in development and troubleshooting, providing valuable information about hand state changes without cluttering release builds.

4.4.2 Custom XR Socket Implementation

The XRSocketTagInteractor extends Unity's standard socket functionality to provide enhanced control over object placement and interaction filtering. This customization addresses specific gameplay requirements while maintaining compatibility with the broader XR Interaction Toolkit ecosystem.

```
public class XRSocketTagInteractor : XRSocketInteractor
{
    [Header("Socket Configuration")]
    public string targetTag = "";

    [Header("Visual Feedback")]
    public Material validHoverMaterial;
    public Material invalidHoverMaterial;
    private Material originalMaterial;
    private Renderer socketRenderer;

    [Header("Audio Feedback")]
    public string validHoverSound = "socket_valid";
    public string invalidHoverSound = "socket_invalid";
    public string insertionSound = "socket_insert";

    protected override void Awake()
    {
        base.Awake();
        socketRenderer = GetComponent<Renderer>();
        if (socketRenderer != null)
        {
            originalMaterial = socketRenderer.material;
        }
    }

    public override bool CanHover(IXRHoverInteractable interactable)
    {
        bool baseCanHover = base.CanHover(interactable);
        bool hasCorrectTag = interactable.transform.tag == targetTag;

        // Provide audio feedback for hover attempts
        if (baseCanHover && !hasCorrectTag)
        {
            PlayAudioFeedback(invalidHoverSound);
            UpdateVisualFeedback(false);
        }
        else if (baseCanHover && hasCorrectTag)
        {
            PlayAudioFeedback(validHoverSound);
            UpdateVisualFeedback(true);
        }

        return baseCanHover && hasCorrectTag;
    }

    public override bool CanSelect(IXRSelectInteractable interactable)
    {
        bool baseCanSelect = base.CanSelect(interactable);
        bool hasCorrectTag = interactable.transform.tag == targetTag;

```

```
52          if (baseCanSelect && hasCorrectTag)
53          {
54              PlayAudioFeedback(insertionSound);
55          }
56
57          return baseCanSelect && hasCorrectTag;
58      }
59
60      private void PlayAudioFeedback(string soundName)
61      {
62          if (!string.IsNullOrEmpty(soundName) && AudioManager.instance
    != null)
63          {
64              AudioManager.instance.Play(soundName);
65          }
66      }
67
68      private void UpdateVisualFeedback(bool isValid)
69      {
70          if (socketRenderer == null) return;
71
72          if (isValid && validHoverMaterial != null)
73          {
74              socketRenderer.material = validHoverMaterial;
75          }
76          else if (!isValid && invalidHoverMaterial != null)
77          {
78              socketRenderer.material = invalidHoverMaterial;
79          }
80      }
81
82      protected override void OnHoverExited(HoverExitEventArgs args)
83      {
84          base.OnHoverExited(args);
85
86          // Restore original visual state
87          if (socketRenderer != null && originalMaterial != null)
88          {
89              socketRenderer.material = originalMaterial;
90          }
91      }
92  }
```

Listing 4: Enhanced Socket Interaction with Tag Filtering

Enhanced Filtering Logic:

The custom implementation extends the base XR socket behavior by adding tag-based filtering that prevents incorrect objects from being placed in sockets. This is crucial for puzzle-solving mechanics where specific objects must be placed in designated locations. The dual-check system (base compatibility plus tag matching) ensures both technical compatibility and logical game state consistency.

Multi-Modal Feedback System:

The implementation provides both visual and audio feedback for interaction attempts, creating rich user experiences that guide players toward correct solutions. The material swapping system provides immediate visual feedback about socket compatibility, while the audio system reinforces these cues with appropriate sound effects.

Inheritance and Integration:

By extending XRSocketInteractor rather than reimplementing socket functionality, this component maintains full compatibility with Unity's XR Interaction Toolkit while adding specialized behavior. This approach demonstrates proper object-oriented design principles and ensures future compatibility with toolkit updates.

4.4.3 Automatic Affordance System

The automatic affordance system streamlines the setup process for interactive objects by automatically establishing relationships between interactable components and their affordance providers.

```
public class AutoFindInteractableAffordance : MonoBehaviour
{
    [Header("Configuration")]
    public bool searchInParent = true;
    public bool searchInChildren = false;
    public bool debugOutput = false;

    private void Awake()
    {
        ConfigureAffordanceProvider();
    }

    private void ConfigureAffordanceProvider()
    {
        // Get the affordance state provider component
        XRInteractableAffordanceStateProvider stateProvider =
            GetComponent<XRInteractableAffordanceStateProvider>();

        if (stateProvider == null)
        {
            if (debugOutput)
            {
                Debug.LogWarning($"No
    XRInteractableAffordanceStateProvider found on {gameObject.name}");
            }
            return;
        }

        // Find the appropriate interactable source
        XRBaseInteractable interactableSource = FindInteractableSource
    ();

        if (interactableSource != null)
        {
            stateProvider.interactableSource = interactableSource;

            if (debugOutput)
            {
                Debug.Log($"Affordance provider on {gameObject.name} "
    +
                          $"configured with interactable: {
    interactableSource.gameObject.name}");
            }
        }
        else
```

```
42          {
43              Debug.LogError($"No suitable XRBaseInteractable found for
    affordance provider on {gameObject.name}");
44          }
45      }
46
47      private XRBaseInteractable FindInteractableSource()
48      {
49          XRBaseInteractable interactable = null;
50
51          // First, try to find on the same GameObject
52          interactable = GetComponent<XRBaseInteractable>();
53          if (interactable != null) return interactable;
54
55          // Search in parent if enabled
56          if (searchInParent)
57          {
58              interactable = GetComponentInParent<XRBaseInteractable>();
59              if (interactable != null) return interactable;
60          }
61
62          // Search in children if enabled
63          if (searchInChildren)
64          {
65              interactable = GetComponentInChildren<XRBaseInteractable>()
    ;
66              if (interactable != null) return interactable;
67          }
68
69          return null;
70      }
71  }
```

Listing 5: Automated Interactable Affordance Configuration

Automated Configuration Benefits:

This utility component eliminates manual configuration steps that are prone to human error and time-consuming during rapid prototyping phases. By automatically establishing component relationships during the Awake phase, the system ensures that affordance providers are properly configured before any interaction events occur.

Flexible Search Strategies:

The configurable search options (same object, parent, children) accommodate various GameObject hierarchies and organizational patterns. This flexibility is particularly valuable in prefab-based development workflows where different organizational approaches may be used for different types of interactive objects.

Development Workflow Integration:

The debug output functionality aids in development and troubleshooting by providing clear information about the configuration process. This is particularly valuable when working with complex prefab hierarchies where component relationships might not be immediately obvious.

4.5 Weapon and Tool Systems

The weapon and tool systems in Space Survival demonstrate advanced VR interaction concepts while providing engaging gameplay mechanics. These systems integrate physics

simulation, visual effects, and audio feedback to create compelling interaction experiences.

4.5.1 Meteor Pistol Implementation

Figure 13: Meteor Pistol in-game view

The MeteorPistol represents a sophisticated approach to VR tool design that combines traditional shooting mechanics with VR-specific interaction paradigms.

```
public class MeteorPistol : MonoBehaviour
{
    [Header("Visual Effects")]
    public ParticleSystem particles;
    public LineRenderer laserLine;
    public Transform muzzleFlash;

    [Header("Physics Configuration")]
    public LayerMask layerMask;
    public Transform shootSource;
    public float distance = 10f;
    public float damageAmount = 1f;

    [Header("Audio Configuration")]
    public string fireStartSound = "Pistol";
    public string fireLoopSound = "PistolLoop";
    public string fireEndSound = "PistolEnd";
    public string hitSound = "MetalHit";

    [Header("Haptic Feedback")]
    public float hapticIntensity = 0.3f;
    public float hapticDuration = 0.1f;

    [Header("Performance Settings")]
    public int raycastsPerFrame = 1;
    public float visualUpdateInterval = 0.02f;

    private bool rayActivate = false;
    private XRGrabInteractable grabInteractable;
    private float lastVisualUpdate = 0f;
    private Coroutine hapticCoroutine;
```

```
32
33      void Start()
34      {
35          InitializeComponents();
36          RegisterInteractionEvents();
37          PrepareVisualElements();
38      }
39
40      private void InitializeComponents()
41      {
42          grabInteractable = GetComponent<XRGrabInteractable>();
43          if (grabInteractable == null)
44          {
45              Debug.LogError($"XRGrabInteractable component required on {
    gameObject.name}");
46              enabled = false;
47              return;
48          }
49
50          if (shootSource == null)
51          {
52              shootSource = transform;
53              Debug.LogWarning($"Shoot source not assigned on {gameObject
    .name}, using transform");
54          }
55      }
56
57      private void RegisterInteractionEvents()
58      {
59          grabInteractable.activated.AddListener(OnActivated);
60          grabInteractable.deactivated.AddListener(OnDeactivated);
61      }
62
63      private void PrepareVisualElements()
64      {
65          if (particles != null)
66          {
67              var main = particles.main;
68              main.loop = true;
69              particles.Stop();
70          }
71
72          if (laserLine != null)
73          {
74              laserLine.enabled = false;
75          }
76      }
77
78      public void OnActivated(ActivateEventArgs args)
79      {
80          StartShoot();
81          ProvideHapticFeedback(args.interactorObject);
82      }
83
84      public void OnDeactivated(DeactivateEventArgs args)
85      {
86          StopShoot();
87      }
```

```
88
89      public void StartShoot()
90      {
91          if (rayActivate) return; // Prevent double activation
92
93          rayActivate = true;
94
95          // Audio feedback
96          if (AudioManager.instance != null)
97          {
98              AudioManager.instance.Play(fireStartSound);
99              if (!string.IsNullOrEmpty(fireLoopSound))
100             {
101                 AudioManager.instance.Play(fireLoopSound);
102             }
103         }
104
105         // Visual effects
106         if (particles != null)
107         {
108             particles.Play();
109         }
110
111         if (muzzleFlash != null)
112         {
113             muzzleFlash.gameObject.SetActive(true);
114         }
115
116         if (laserLine != null)
117         {
118             laserLine.enabled = true;
119         }
120     }
121
122     public void StopShoot()
123     {
124         if (!rayActivate) return;
125
126         rayActivate = false;
127
128         // Stop audio
129         if (AudioManager.instance != null)
130         {
131             if (!string.IsNullOrEmpty(fireLoopSound))
132             {
133                 AudioManager.instance.Stop(fireLoopSound);
134             }
135             AudioManager.instance.Play(fireEndSound);
136         }
137
138         // Stop visual effects
139         if (particles != null)
140         {
141             particles.Stop(true, ParticleSystemStopBehavior.
    StopEmittingAndClear);
142         }
143
144         if (muzzleFlash != null)
```

```
145         {
146             muzzleFlash.gameObject.SetActive(false);
147         }
148 
149         if (laserLine != null)
150         {
151             laserLine.enabled = false;
152         }
153     }
154 
155     void Update()
156     {
157         if (rayActivate)
158         {
159             PerformRaycastDetection();
160             UpdateVisualEffects();
161         }
162     }
163 
164     private void PerformRaycastDetection()
165     {
166         for (int i = 0; i < raycastsPerFrame; i++)
167         {
168             RaycastHit hit;
169             bool hasHit = Physics.Raycast(shootSource.position,
    shootSource.forward,
170                                           out hit, distance, layerMask);
171 
172             if (hasHit)
173             {
174                 ProcessHit(hit);
175             }
176 
177             UpdateLaserVisual(hasHit ? hit.point : shootSource.position
     + shootSource.forward * distance);
178         }
179     }
180 
181     private void ProcessHit(RaycastHit hit)
182     {
183         // Send damage message to hit object
184         hit.transform.gameObject.SendMessage("Break",
    SendMessageOptions.DontRequireReceiver);
185 
186         // Play hit sound
187         if (AudioManager.instance != null && !string.IsNullOrEmpty(
    hitSound))
188         {
189             AudioManager.instance.Play(hitSound);
190         }
191 
192         // Create impact effect at hit point
193         CreateImpactEffect(hit.point, hit.normal);
194     }
195 
196     private void UpdateLaserVisual(Vector3 endPoint)
197     {
198         if (laserLine != null && Time.time - lastVisualUpdate >
```

```
    visualUpdateInterval)
199         {
200             laserLine.SetPosition(0, shootSource.position);
201             laserLine.SetPosition(1, endPoint);
202             lastVisualUpdate = Time.time;
203         }
204     }
205
206     private void CreateImpactEffect(Vector3 position, Vector3 normal)
207     {
208         // Implementation for impact particle effects
209         // This could instantiate impact particles, spark effects, etc.
210     }
211
212     private void ProvideHapticFeedback(IXRSelectInteractor interactor)
213     {
214         if (hapticCoroutine != null)
215         {
216             StopCoroutine(hapticCoroutine);
217         }
218         hapticCoroutine = StartCoroutine(HapticFeedbackRoutine(
    interactor));
219     }
220
221     private IEnumerator HapticFeedbackRoutine(IXRSelectInteractor
    interactor)
222     {
223         XRBaseControllerInteractor controllerInteractor = interactor as
     XRBaseControllerInteractor;
224         if (controllerInteractor != null)
225         {
226             controllerInteractor.SendHapticImpulse(hapticIntensity,
    hapticDuration);
227         }
228         yield return new WaitForSeconds(hapticDuration);
229     }
230 }
```

Listing 6: Advanced VR Tool with Raycast Detection and Visual Effects

Advanced Raycast Implementation:

The raycast system demonstrates sophisticated understanding of VR physics requirements. Unlike traditional shooting mechanics that might use instantaneous hit detection, this implementation considers VR-specific factors such as hand stability and the need for immediate visual feedback. The configurable raycast frequency allows for performance tuning while maintaining responsive interaction.

Multi-Modal Feedback Integration:

The tool provides comprehensive feedback through visual, audio, and haptic channels. The particle system creates immediate visual confirmation of tool activation, while the laser line renderer provides continuous visual feedback about the tool's aim direction. The haptic feedback integration adds tactile confirmation that enhances the physical sense of tool operation.

Performance Optimization Considerations:

The implementation includes several performance optimization strategies essential for VR applications. The raycast frequency limitation prevents excessive physics calcu-

lations, while the visual update interval prevents unnecessary rendering updates. These optimizations ensure that the tool can operate smoothly within the demanding performance requirements of VR applications.

State Management and Safety:

The comprehensive state management prevents common issues such as double-activation or resource leaks. The activation state tracking ensures that audio and visual effects are properly managed throughout the tool's lifecycle, preventing audio conflicts or visual artifacts that could disrupt the VR experience.

4.5.2 Breakable Object System

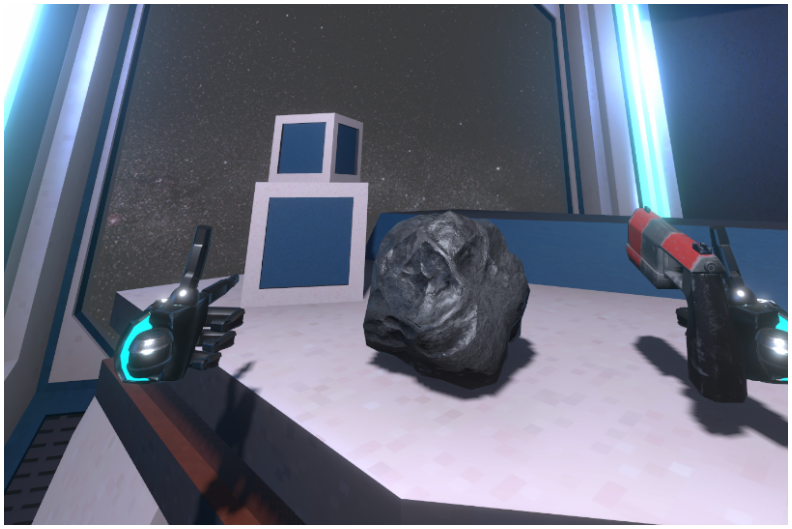

Figure 14: Breakable Meteor before destruction

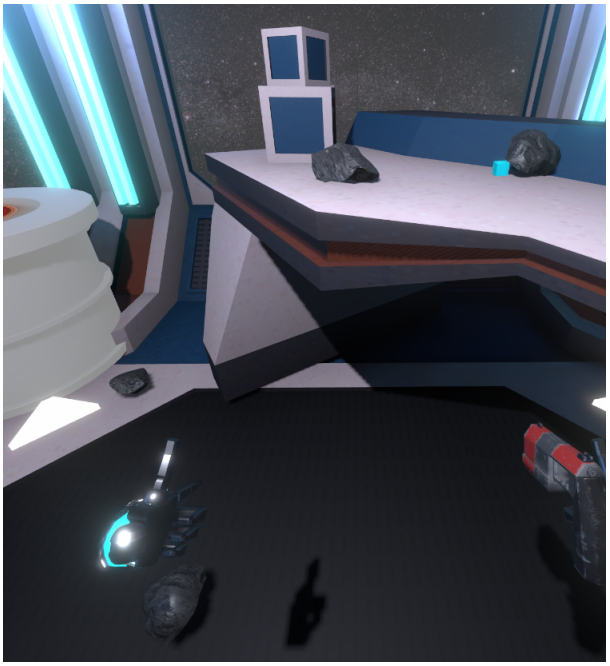

Figure 15: Breakable Meteor after destruction, showing fragments

The Breakable system provides dynamic environmental destruction that responds to tool interactions while maintaining performance efficiency necessary for VR applications.

```
public class Breakable : MonoBehaviour
{
    [Header("Destruction Configuration")]
    public List<GameObject> Breakables = new List<GameObject>();
    public float timeToBreak = 2f;
    public bool requiresContinuousInput = true;
    public float inputResetDelay = 0.5f;

    [Header("Visual Feedback")]
    public Material damageMaterial;
    public ParticleSystem breakEffect;
    public float damageEffectThreshold = 0.5f;

    [Header("Audio Configuration")]
    public string damageSound = "MetalStrain";
    public string breakSound = "MetalBreak";
    public float audioFadeInTime = 0.2f;

    [Header("Physics Behavior")]
    public float explosionForce = 100f;
    public float explosionRadius = 2f;
    public Vector3 explosionOffset = Vector3.zero;

    [Header("Events")]
    public UnityEvent OnDamageStart;
    public UnityEvent<float> OnDamageProgress;
    public UnityEvent OnBreak;

    private float accumulatedDamage = 0f;
    private bool isBroken = false;
    private float lastDamageTime = 0f;
    private Renderer objectRenderer;
    private Material originalMaterial;
    private AudioSource damageAudioSource;
    private Coroutine damageCoroutine;

    void Start()
    {
        InitializeComponents();
        PrepareFragments();
        ValidateConfiguration();
    }

    private void InitializeComponents()
    {
        objectRenderer = GetComponent<Renderer>();
        if (objectRenderer != null)
        {
            originalMaterial = objectRenderer.material;
        }

        // Setup audio source for damage sounds
        damageAudioSource = GetComponent<AudioSource>();
        if (damageAudioSource == null && !string.IsNullOrEmpty(
   damageSound))
```

```
55          {
56              damageAudioSource = gameObject.AddComponent<AudioSource>();
57              damageAudioSource.loop = true;
58              damageAudioSource.volume = 0f;
59          }
60      }
61
62      private void PrepareFragments()
63      {
64          foreach (var fragment in Breakables)
65          {
66              if (fragment != null)
67              {
68                  fragment.SetActive(false);
69
70                  // Ensure fragments have proper physics components
71                  Rigidbody rb = fragment.GetComponent<Rigidbody>();
72                  if (rb == null)
73                  {
74                      rb = fragment.AddComponent<Rigidbody>();
75                  }
76                  rb.isKinematic = true; // Start as kinematic, enable
    physics on break
77
78                  // Add cleanup component
79                  FragmentCleanup cleanup = fragment.GetComponent<
    FragmentCleanup>();
80                  if (cleanup == null)
81                  {
82                      cleanup = fragment.AddComponent<FragmentCleanup>();
83                      cleanup.lifetime = 10f; // Auto-cleanup after 10
    seconds
84                  }
85              }
86          }
87      }
88
89      private void ValidateConfiguration()
90      {
91          if (Breakables.Count == 0)
92          {
93              Debug.LogWarning($"No breakable fragments assigned to {
    gameObject.name}");
94          }
95
96          if (timeToBreak <= 0)
97          {
98              Debug.LogError($"Invalid timeToBreak value on {gameObject.
    name}. Must be greater than 0.");
99              timeToBreak = 1f;
100         }
101     }
102
103     public void Break()
104     {
105         if (isBroken) return;
106
107         if (requiresContinuousInput)
```

```
108         {
109             ProcessContinuousDamage();
110         }
111         else
112         {
113             ProcessInstantBreak();
114         }
115     }
116 
117     private void ProcessContinuousDamage()
118     {
119         float currentTime = Time.time;
120 
121         // Reset damage if too much time has passed since last input
122         if (currentTime - lastDamageTime > inputResetDelay)
123         {
124             if (accumulatedDamage > 0)
125             {
126                 StartDamageRecovery();
127             }
128         }
129 
130         lastDamageTime = currentTime;
131         accumulatedDamage += Time.deltaTime;
132 
133         // Start damage effects if threshold reached
134         if (accumulatedDamage >= damageEffectThreshold * timeToBreak &&
     damageCoroutine == null)
135         {
136             damageCoroutine = StartCoroutine(DamageEffectSequence());
137             OnDamageStart.Invoke();
138         }
139 
140         // Update damage progress
141         float damageProgress = Mathf.Clamp01(accumulatedDamage /
    timeToBreak);
142         OnDamageProgress.Invoke(damageProgress);
143 
144         // Check if object should break
145         if (accumulatedDamage >= timeToBreak)
146         {
147             ExecuteBreak();
148         }
149     }
150 
151     private void ProcessInstantBreak()
152     {
153         ExecuteBreak();
154     }
155 
156     private IEnumerator DamageEffectSequence()
157     {
158         // Start damage audio
159         if (damageAudioSource != null && !string.IsNullOrEmpty(
    damageSound) && AudioManager.instance != null)
160         {
161             AudioManager.instance.Play(damageSound);
162 
```

```
163             // Fade in damage audio
164             float fadeTimer = 0f;
165             while (fadeTimer < audioFadeInTime)
166             {
167                 fadeTimer += Time.deltaTime;
168                 float fadeProgress = fadeTimer / audioFadeInTime;
169                 damageAudioSource.volume = fadeProgress * 0.7f; //
    Target volume
170                 yield return null;
171             }
172         }
173 
174         // Apply damage visual effects
175         if (objectRenderer != null && damageMaterial != null)
176         {
177             float flashTimer = 0f;
178             while (!isBroken && accumulatedDamage >
    damageEffectThreshold * timeToBreak)
179             {
180                 // Flash between original and damage material
181                 objectRenderer.material = (Mathf.Sin(flashTimer * 10f)
    > 0) ? damageMaterial : originalMaterial;
182                 flashTimer += Time.deltaTime;
183                 yield return null;
184             }
185 
186             // Restore original material if not broken
187             if (!isBroken)
188             {
189                 objectRenderer.material = originalMaterial;
190             }
191         }
192     }
193 
194     private void StartDamageRecovery()
195     {
196         StartCoroutine(DamageRecoverySequence());
197     }
198 
199     private IEnumerator DamageRecoverySequence()
200     {
201         float startDamage = accumulatedDamage;
202         float recoveryTime = 1f; // Time to fully recover
203         float timer = 0f;
204 
205         while (timer < recoveryTime && Time.time - lastDamageTime >
    inputResetDelay)
206         {
207             timer += Time.deltaTime;
208             float recoveryProgress = timer / recoveryTime;
209             accumulatedDamage = Mathf.Lerp(startDamage, 0f,
    recoveryProgress);
210 
211             OnDamageProgress.Invoke(accumulatedDamage / timeToBreak);
212             yield return null;
213         }
214 
215         if (damageCoroutine != null)
```

```
216         {
217             StopCoroutine(damageCoroutine);
218             damageCoroutine = null;
219         }
220 
221         // Restore original appearance
222         if (objectRenderer != null)
223         {
224             objectRenderer.material = originalMaterial;
225         }
226     }
227 
228     private void ExecuteBreak()
229     {
230         if (isBroken) return;
231         isBroken = true;
232 
233         // Stop damage effects
234         if (damageCoroutine != null)
235         {
236             StopCoroutine(damageCoroutine);
237         }
238 
239         // Play break audio
240         if (AudioManager.instance != null && !string.IsNullOrEmpty(
    breakSound))
241         {
242             AudioManager.instance.Play(breakSound);
243         }
244 
245         // Stop damage audio
246         if (damageAudioSource != null)
247         {
248             damageAudioSource.Stop();
249         }
250 
251         // Create break particle effect
252         if (breakEffect != null)
253         {
254             Instantiate(breakEffect, transform.position, transform.
    rotation);
255         }
256 
257         // Activate and launch fragments
258         ActivateFragments();
259 
260         // Invoke break events
261         OnBreak.Invoke();
262 
263         // Deactivate original object
264         gameObject.SetActive(false);
265     }
266 
267     private void ActivateFragments()
268     {
269         Vector3 explosionCenter = transform.position + explosionOffset;
270 
271         foreach (var fragment in Breakables)
```

```
272         {
273             if (fragment != null)
274             {
275                 // Position fragment at original object location
276                 fragment.transform.position = transform.position +
    Random.insideUnitSphere * 0.1f;
277                 fragment.transform.rotation = transform.rotation *
    Quaternion.Euler(Random.insideUnitSphere * 30f);
278 
279                 fragment.SetActive(true);
280                 fragment.transform.parent = null; // Unparent for
    independent physics
281 
282                 // Enable physics and apply explosion force
283                 Rigidbody rb = fragment.GetComponent<Rigidbody>();
284                 if (rb != null)
285                 {
286                     rb.isKinematic = false;
287                     rb.AddExplosionForce(explosionForce,
    explosionCenter, explosionRadius);
288 
289                     // Add random torque for realistic tumbling
290                     rb.AddTorque(Random.insideUnitSphere *
    explosionForce * 0.1f);
291                 }
292             }
293         }
294     }
295 
296     void Update()
297     {
298         if (!isBroken && requiresContinuousInput && accumulatedDamage >
    0)
299         {
300             // Handle damage decay when not receiving input
301             if (Time.time - lastDamageTime > inputResetDelay)
302             {
303                 if (damageCoroutine == null)
304                 {
305                     StartDamageRecovery();
306                 }
307             }
308         }
309     }
310 }
311 
312 // Supporting class for fragment cleanup
313 public class FragmentCleanup : MonoBehaviour
314 {
315     public float lifetime = 10f;
316 
317     void Start()
318     {
319         Destroy(gameObject, lifetime);
320     }
321 }
```

Listing 7: Advanced Destructible Environment System

Progressive Damage System:

The breakable system implements sophisticated damage accumulation that requires sustained tool interaction rather than instant destruction. This design choice enhances gameplay depth while providing multiple feedback opportunities to guide player behavior. The continuous input requirement creates more engaging tool usage patterns that feel natural in VR environments.

Multi-Stage Visual Feedback:

The system provides progressive visual feedback through material changes, particle effects, and audio cues that communicate damage state to players. This multi-modal approach ensures that players receive clear information about object status without relying solely on UI elements that might break immersion in VR.

Physics-Based Fragment System:

The fragment activation system demonstrates advanced understanding of Unity's physics system and VR performance requirements. Fragments are pre-configured but kept inactive until needed, preventing performance overhead from unused physics objects. The explosion force application creates realistic debris behavior that enhances the satisfaction of environmental destruction.

Memory and Performance Management:

The fragment cleanup system prevents memory leaks that could accumulate during extended gameplay sessions. The automatic cleanup timing balances visual persistence with memory management, ensuring that debris remains visible long enough to provide satisfactory feedback while preventing performance degradation from excessive physics objects.

4.6 Environmental Interaction Systems

Environmental interaction systems create the dynamic world elements that players interact with throughout the Space Survival experience. These systems balance complexity with performance while providing meaningful feedback for player actions.

4.6.1 Trigger Zone Implementation

The TriggerZone system provides a flexible foundation for environmental interactions that can be configured for diverse gameplay scenarios.

```
public class TriggerZone : MonoBehaviour
{
    [Header("Trigger Configuration")]
    public string targetTag = "";
    public bool allowMultipleObjects = false;
    public bool requireVelocityThreshold = false;
    public float minimumVelocity = 1f;

    [Header("Response Behavior")]
    public float responseDelay = 0f;
    public bool singleUse = false;
    public bool requireObjectToStop = false;
    public float stopVelocityThreshold = 0.1f;

    [Header("Visual Feedback")]
    public GameObject activationEffect;
    public Material triggerMaterial;
```

```
18      public bool showDebugBounds = false;
19
20      [Header("Audio Configuration")]
21      public string enterSound = "";
22      public string exitSound = "";
23      public string activationSound = "";
24
25      [Header("Events")]
26      public UnityEvent<GameObject> OnEnterEvent;
27      public UnityEvent<GameObject> OnExitEvent;
28      public UnityEvent<GameObject> OnActivationEvent;
29
30      private HashSet<GameObject> objectsInZone = new HashSet<GameObject
        >();
31      private bool hasBeenUsed = false;
32      private Renderer zoneRenderer;
33      private Material originalMaterial;
34      private Coroutine delayedResponseCoroutine;
35
36      void Start()
37      {
38          InitializeVisualComponents();
39          ValidateConfiguration();
40          SetupColliderRequirements();
41      }
42
43      private void InitializeVisualComponents()
44      {
45          zoneRenderer = GetComponent<Renderer>();
46          if (zoneRenderer != null)
47          {
48              originalMaterial = zoneRenderer.material;
49
50              if (!showDebugBounds && Application.isPlaying)
51              {
52                  zoneRenderer.enabled = false;
53              }
54          }
55      }
56
57      private void ValidateConfiguration()
58      {
59          Collider trigger = GetComponent<Collider>();
60          if (trigger == null)
61          {
62              Debug.LogError($"TriggerZone on {gameObject.name} requires
        a Collider component");
63              return;
64          }
65
66          if (!trigger.isTrigger)
67          {
68              Debug.LogWarning($"Collider on {gameObject.name} should be
        set as trigger");
69              trigger.isTrigger = true;
70          }
71
72          if (string.IsNullOrEmpty(targetTag))
```

```
        {
            Debug.LogWarning($"No target tag specified for TriggerZone on {gameObject.name}");
        }
    }

    private void SetupColliderRequirements()
    {
        // Ensure the GameObject is on an appropriate layer for trigger detection
        if (gameObject.layer == 0) // Default layer
        {
            Debug.LogWarning($"TriggerZone on {gameObject.name} is on Default layer. Consider using a dedicated trigger layer.");
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if (hasBeenUsed && singleUse) return;

        if (IsValidTarget(other.gameObject))
        {
            HandleObjectEnter(other.gameObject);
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (objectsInZone.Contains(other.gameObject))
        {
            HandleObjectExit(other.gameObject);
        }
    }

    private bool IsValidTarget(GameObject obj)
    {
        // Check tag matching
        if (!string.IsNullOrEmpty(targetTag) && obj.tag != targetTag)
        {
            return false;
        }

        // Check velocity requirements
        if (requireVelocityThreshold)
        {
            Rigidbody rb = obj.GetComponent<Rigidbody>();
            if (rb == null || rb.velocity.magnitude < minimumVelocity)
            {
                return false;
            }
        }

        // Check multiple object restrictions
        if (!allowMultipleObjects && objectsInZone.Count > 0)
        {
            return false;
        }
```

```
128
129         return true;
130     }
131
132     private void HandleObjectEnter(GameObject obj)
133     {
134         if (objectsInZone.Add(obj)) // Returns true if object was added
    (wasn't already in set)
135         {
136             // Play enter sound
137             if (!string.IsNullOrEmpty(enterSound) && AudioManager.
    instance != null)
138             {
139                 AudioManager.instance.Play(enterSound);
140             }
141
142             // Update visual state
143             UpdateVisualState(true);
144
145             // Invoke enter event
146             OnEnterEvent.Invoke(obj);
147
148             // Handle activation logic
149             if (requireObjectToStop)
150             {
151                 StartCoroutine(MonitorObjectForStop(obj));
152             }
153             else if (responseDelay > 0)
154             {
155                 if (delayedResponseCoroutine != null)
156                 {
157                     StopCoroutine(delayedResponseCoroutine);
158                 }
159                 delayedResponseCoroutine = StartCoroutine(
    DelayedActivation(obj));
160             }
161             else
162             {
163                 ActivateTrigger(obj);
164             }
165         }
166     }
167
168     private void HandleObjectExit(GameObject obj)
169     {
170         if (objectsInZone.Remove(obj))
171         {
172             // Play exit sound
173             if (!string.IsNullOrEmpty(exitSound) && AudioManager.
    instance != null)
174             {
175                 AudioManager.instance.Play(exitSound);
176             }
177
178             // Update visual state
179             if (objectsInZone.Count == 0)
180             {
181                 UpdateVisualState(false);
```

```
            }

            // Invoke exit event
            OnExitEvent.Invoke(obj);
        }
    }

    private IEnumerator MonitorObjectForStop(GameObject obj)
    {
        Rigidbody rb = obj.GetComponent<Rigidbody>();
        if (rb == null) yield break;

        // Wait for object to come to a stop
        while (objectsInZone.Contains(obj) && rb.velocity.magnitude > stopVelocityThreshold)
        {
            yield return new WaitForFixedUpdate();
        }

        // Activate if object is still in zone and has stopped
        if (objectsInZone.Contains(obj))
        {
            if (responseDelay > 0)
            {
                yield return new WaitForSeconds(responseDelay);
            }

            if (objectsInZone.Contains(obj)) // Double-check object is still present
            {
                ActivateTrigger(obj);
            }
        }
    }

    private IEnumerator DelayedActivation(GameObject obj)
    {
        yield return new WaitForSeconds(responseDelay);

        if (objectsInZone.Contains(obj))
        {
            ActivateTrigger(obj);
        }
    }

    private void ActivateTrigger(GameObject obj)
    {
        // Play activation sound
        if (!string.IsNullOrEmpty(activationSound) && AudioManager.instance != null)
        {
            AudioManager.instance.Play(activationSound);
        }

        // Create activation effect
        if (activationEffect != null)
        {
            Instantiate(activationEffect, transform.position, transform
```

```
    .rotation);
        }

        // Invoke activation event
        OnActivationEvent.Invoke(obj);

        // Mark as used if single-use
        if (singleUse)
        {
            hasBeenUsed = true;
            UpdateVisualState(false);
        }
    }

    private void UpdateVisualState(bool active)
    {
        if (zoneRenderer != null && showDebugBounds)
        {
            if (active && triggerMaterial != null)
            {
                zoneRenderer.material = triggerMaterial;
            }
            else if (originalMaterial != null)
            {
                zoneRenderer.material = originalMaterial;
            }
        }
    }

    void OnDrawGizmosSelected()
    {
        // Draw trigger bounds in scene view
        Collider col = GetComponent<Collider>();
        if (col != null)
        {
            Gizmos.color = hasBeenUsed ? Color.red : Color.green;
            Gizmos.matrix = transform.localToWorldMatrix;

            if (col is BoxCollider)
            {
                BoxCollider box = col as BoxCollider;
                Gizmos.DrawWireCube(box.center, box.size);
            }
            else if (col is SphereCollider)
            {
                SphereCollider sphere = col as SphereCollider;
                Gizmos.DrawWireSphere(sphere.center, sphere.radius);
            }
        }
    }
}
```

Listing 8: Flexible Environmental Trigger System

Advanced Configuration Options:

The TriggerZone system provides extensive configuration options that accommodate diverse gameplay requirements. The velocity threshold feature enables physics-based interactions where objects must be thrown with sufficient force, while the stop detection

system supports placement-based puzzles where objects must come to rest within the zone.

State Management and Safety:

The HashSet-based object tracking ensures efficient membership testing while preventing duplicate event firing. The single-use functionality supports progression mechanics where triggers should only activate once, while the multiple object restrictions enable exclusive interaction requirements.

Performance Optimization:

The coroutine-based monitoring systems minimize performance impact by using appropriate wait conditions rather than continuous Update polling. The gizmo visualization aids in development while remaining editor-only to prevent runtime performance impact.

4.6.2 Waste Management System Implementation

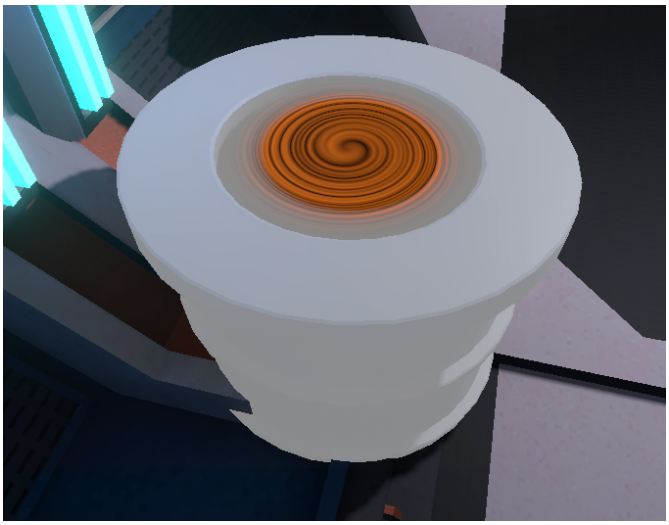

Figure 16: In-game trash can with disposal animation and effect

Figure 17: TrashCan Hierarchy in Unity

The TrashCan system demonstrates specialized TriggerZone usage for environmental cleanup mechanics that maintain scene performance and provide satisfying interaction feedback.

```
public class TrashCan : MonoBehaviour
{
    [Header("Disposal Configuration")]
    public List<string> acceptedTags = new List<string> { "Debris", "
   Trash", "Disposable" };
    public bool requireCorrectDisposal = true;
    public float disposalDelay = 0.5f;

    [Header("Capacity Configuration")]
```

```
    public int maxCapacity = -1; // -1 for unlimited
    public bool showCapacityWarning = true;
    public string capacityWarningSound = "ContainerFull";

    [Header("Visual Feedback")]
    public ParticleSystem disposalEffect;
    public Animation lidAnimation;
    public Material fullMaterial;
    public Transform fillIndicator;

    [Header("Audio Feedback")]
    public string successDisposalSound = "TrashDisposal";
    public string rejectionSound = "ErrorBeep";
    public string capacitySound = "ContainerClink";

    [Header("Performance Settings")]
    public bool enableObjectPooling = true;
    public float poolReturnDelay = 2f;

    private int currentLoad = 0;
    private TriggerZone triggerZone;
    private bool isAtCapacity = false;
    private Queue<GameObject> disposalQueue = new Queue<GameObject>();
    private Coroutine disposalProcessCoroutine;

    // Object pooling support
    private Dictionary<string, Queue<GameObject>> objectPools =
        new Dictionary<string, Queue<GameObject>>();

    void Start()
    {
        InitializeComponents();
        SetupTriggerEvents();
        InitializeObjectPooling();
        UpdateVisualState();
    }

    private void InitializeComponents()
    {
        triggerZone = GetComponent<TriggerZone>();
        if (triggerZone == null)
        {
            Debug.LogError($"TrashCan on {gameObject.name} requires a
TriggerZone component");
            enabled = false;
            return;
        }

        // Configure trigger zone for disposal behavior
        triggerZone.allowMultipleObjects = true;
        triggerZone.singleUse = false;
    }

    private void SetupTriggerEvents()
    {
        triggerZone.OnEnterEvent.RemoveAllListeners();
        triggerZone.OnEnterEvent.AddListener(AttemptDisposal);
    }
```

```
66
67      private void InitializeObjectPooling()
68      {
69          if (!enableObjectPooling) return;
70
71          foreach (string tag in acceptedTags)
72          {
73              objectPools[tag] = new Queue<GameObject>();
74          }
75      }
76
77      public void AttemptDisposal(GameObject obj)
78      {
79          if (isAtCapacity)
80          {
81              HandleCapacityExceeded(obj);
82              return;
83          }
84
85          if (IsValidDisposal(obj))
86          {
87              QueueForDisposal(obj);
88          }
89          else
90          {
91              HandleInvalidDisposal(obj);
92          }
93      }
94
95      private bool IsValidDisposal(GameObject obj)
96      {
97          if (!requireCorrectDisposal) return true;
98
99          return acceptedTags.Contains(obj.tag);
100     }
101
102     private void QueueForDisposal(GameObject obj)
103     {
104         disposalQueue.Enqueue(obj);
105
106         if (disposalProcessCoroutine == null)
107         {
108             disposalProcessCoroutine = StartCoroutine(
    ProcessDisposalQueue());
109         }
110     }
111
112     private IEnumerator ProcessDisposalQueue()
113     {
114         while (disposalQueue.Count > 0 && !isAtCapacity)
115         {
116             GameObject obj = disposalQueue.Dequeue();
117
118             if (obj != null && obj.activeInHierarchy)
119             {
120                 yield return StartCoroutine(DisposeObject(obj));
121             }
122         }
```

```
123
124         disposalProcessCoroutine = null;
125     }
126
127     private IEnumerator DisposeObject(GameObject obj)
128     {
129         // Play intake animation
130         if (lidAnimation != null)
131         {
132             lidAnimation.Play("LidOpen");
133         }
134
135         // Play capacity sound if container is getting full
136         if (maxCapacity > 0 && currentLoad >= maxCapacity * 0.8f)
137         {
138             if (AudioManager.instance != null && !string.IsNullOrEmpty(
    capacitySound))
139             {
140                 AudioManager.instance.Play(capacitySound);
141             }
142         }
143
144         // Wait for disposal delay
145         yield return new WaitForSeconds(disposalDelay);
146
147         // Create disposal effect
148         if (disposalEffect != null)
149         {
150             disposalEffect.Play();
151         }
152
153         // Play success sound
154         if (AudioManager.instance != null && !string.IsNullOrEmpty(
    successDisposalSound))
155         {
156             AudioManager.instance.Play(successDisposalSound);
157         }
158
159         // Handle object disposal
160         HandleSuccessfulDisposal(obj);
161
162         // Close lid animation
163         if (lidAnimation != null)
164         {
165             lidAnimation.Play("LidClose");
166         }
167
168         // Update capacity
169         currentLoad++;
170         UpdateCapacityState();
171         UpdateVisualState();
172     }
173
174     private void HandleSuccessfulDisposal(GameObject obj)
175     {
176         if (enableObjectPooling)
177         {
178             ReturnToPool(obj);
```

```
179         }
180         else
181         {
182             obj.SetActive(false);
183         }
184     }
185 
186     private void ReturnToPool(GameObject obj)
187     {
188         string objTag = obj.tag;
189 
190         if (objectPools.ContainsKey(objTag))
191         {
192             // Reset object state
193             Rigidbody rb = obj.GetComponent<Rigidbody>();
194             if (rb != null)
195             {
196                 rb.velocity = Vector3.zero;
197                 rb.angularVelocity = Vector3.zero;
198             }
199 
200             // Deactivate and pool
201             obj.SetActive(false);
202             objectPools[objTag].Enqueue(obj);
203         }
204         else
205         {
206             obj.SetActive(false);
207         }
208     }
209 
210     private void HandleInvalidDisposal(GameObject obj)
211     {
212         // Play rejection sound
213         if (AudioManager.instance != null && !string.IsNullOrEmpty(
    rejectionSound))
214         {
215             AudioManager.instance.Play(rejectionSound);
216         }
217 
218         // Add rejection force to push object away
219         Rigidbody rb = obj.GetComponent<Rigidbody>();
220         if (rb != null)
221         {
222             Vector3 rejectionDirection = (obj.transform.position -
    transform.position).normalized;
223             rb.AddForce(rejectionDirection * 5f, ForceMode.Impulse);
224         }
225 
226         Debug.LogWarning($"Invalid disposal attempt: {obj.name} (tag: {
    obj.tag}) not accepted by {gameObject.name}");
227     }
228 
229     private void HandleCapacityExceeded(GameObject obj)
230     {
231         if (showCapacityWarning && AudioManager.instance != null && !
    string.IsNullOrEmpty(capacityWarningSound))
232         {
```

```
233             AudioManager.instance.Play(capacityWarningSound);
234         }
235 
236         // Bounce object away
237         Rigidbody rb = obj.GetComponent<Rigidbody>();
238         if (rb != null)
239         {
240             Vector3 bounceDirection = (obj.transform.position -
    transform.position).normalized;
241             rb.AddForce(bounceDirection * 3f + Vector3.up * 2f,
    ForceMode.Impulse);
242         }
243     }
244 
245     private void UpdateCapacityState()
246     {
247         if (maxCapacity > 0)
248         {
249             isAtCapacity = currentLoad >= maxCapacity;
250         }
251     }
252 
253     private void UpdateVisualState()
254     {
255         // Update fill indicator
256         if (fillIndicator != null && maxCapacity > 0)
257         {
258             float fillPercent = (float)currentLoad / maxCapacity;
259             fillIndicator.localScale = new Vector3(1f, fillPercent, 1f)
    ;
260         }
261 
262         // Update material if at capacity
263         Renderer renderer = GetComponent<Renderer>();
264         if (renderer != null && fullMaterial != null && isAtCapacity)
265         {
266             renderer.material = fullMaterial;
267         }
268     }
269 
270     public GameObject GetPooledObject(string tag)
271     {
272         if (!enableObjectPooling || !objectPools.ContainsKey(tag) ||
    objectPools[tag].Count == 0)
273         {
274             return null;
275         }
276 
277         GameObject pooledObj = objectPools[tag].Dequeue();
278         pooledObj.SetActive(true);
279         return pooledObj;
280     }
281 
282     public void EmptyContainer()
283     {
284         currentLoad = 0;
285         isAtCapacity = false;
286         UpdateVisualState();
```

```
287
288         // Clear disposal queue
289         disposalQueue.Clear();
290
291         // Play empty sound if available
292         if (AudioManager.instance != null)
293         {
294             AudioManager.instance.Play("ContainerEmpty");
295         }
296     }
297
298     // Debug information
299     void OnGUI()
300     {
301         if (!Application.isPlaying || !showCapacityWarning) return;
302
303         if (maxCapacity > 0)
304         {
305             Vector3 screenPos = Camera.main.WorldToScreenPoint(
    transform.position + Vector3.up * 2f);
306             if (screenPos.z > 0)
307             {
308                 GUI.Label(new Rect(screenPos.x - 50, Screen.height -
    screenPos.y, 100, 20),
309                           $"Capacity: {currentLoad}/{maxCapacity}");
310             }
311         }
312     }
313 }
```

Listing 9: Specialized Environmental Disposal System

Sophisticated Capacity Management:

The TrashCan system demonstrates advanced resource management through capacity tracking, visual feedback, and appropriate rejection behaviors. The capacity system prevents infinite object disposal while providing clear feedback about container state, supporting both gameplay mechanics and performance management.

Object Pooling Integration:

The object pooling system addresses VR performance requirements by reusing disposed objects rather than continuously creating and destroying GameObjects. This approach prevents garbage collection spikes that could cause frame rate drops in VR environments while maintaining the illusion of object disposal.

Multi-Modal Feedback System:

The system provides comprehensive feedback through visual animations, particle effects, audio cues, and physical object responses. This multi-modal approach ensures that players receive clear information about disposal success or failure, supporting both accessibility and immersion requirements.

Queue-Based Processing:

The disposal queue system prevents simultaneous disposal conflicts while maintaining responsive interaction feedback. The coroutine-based processing enables smooth disposal animations without blocking other game systems, which is particularly important for maintaining VR frame rates.

4.7 Navigation and Movement Systems

Navigation systems in VR require careful balance between user agency, comfort, and immersion. The Space Survival navigation systems demonstrate multiple approaches to VR movement that accommodate different user preferences and comfort levels.

4.7.1 Spaceship Control System

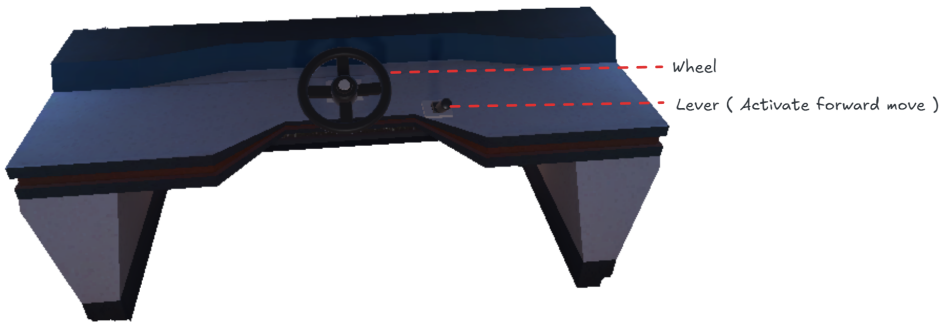

Figure 18: Spaceship cockpit control panel in VR

The SpaceOutsideController represents a sophisticated approach to vehicle control in VR that provides intuitive interaction while maintaining user comfort.

```
public class SpaceOutsideController : MonoBehaviour
{
    [Header("Control Inputs")]
    public XRLever lever;
    public XRKnob knob;
    public XRButton emergencyStop;

    [Header("Movement Configuration")]
    public float forwardSpeed = 5f;
    public float sideSpeed = 3f;
    public float rotationSpeed = 30f;
    public bool enableRotation = false;

    [Header("Physics Behavior")]
    public float acceleration = 2f;
    public float deceleration = 3f;
    public float maxVelocity = 10f;
    public AnimationCurve speedCurve = AnimationCurve.EaseInOut(0, 0,
   1, 1);

    [Header("Audio Configuration")]
    public string engineStartSound = "EngineStart";
    public string engineLoopSound = "Engine";
    public string engineStopSound = "EngineStop";
    public string thusterSound = "SideThrusters";

    [Header("Visual Effects")]
    public ParticleSystem mainEngineEffect;
    public ParticleSystem[] sideThrusters;
    public Light engineLight;
```

```
30      public Material activeMaterial;
31      public Material inactiveMaterial;
32
33      [Header("Comfort Settings")]
34      public bool enableComfortVignette = true;
35      public float vignetteIntensity = 0.3f;
36      public float vignetteBlendTime = 0.2f;
37      public Transform comfortVignette;
38
39      [Header("Environment Response")]
40      public Transform asteroidField;
41      public float fieldMovementMultiplier = 0.5f;
42      public bool enableEnvironmentParallax = true;
43
44      private Vector3 currentVelocity = Vector3.zero;
45      private Vector3 targetVelocity = Vector3.zero;
46      private bool engineActive = false;
47      private bool wasEngineActive = false;
48      private AudioSource engineAudioSource;
49      private Coroutine vignetteCoroutine;
50      private float currentKnobValue = 0f;
51      private bool emergencyStopActivated = false;
52
53      void Start()
54      {
55          InitializeComponents();
56          ValidateControlInputs();
57          SetupAudioSources();
58          SetupInitialState();
59          RegisterControlEvents();
60      }
61
62      private void InitializeComponents()
63      {
64          // Create dedicated audio source for engine sounds
65          engineAudioSource = gameObject.AddComponent<AudioSource>();
66          engineAudioSource.loop = true;
67          engineAudioSource.volume = 0f;
68          engineAudioSource.spatialBlend = 0f; // 2D sound for better
    control
69
70          // Initialize comfort vignette
71          if (enableComfortVignette && comfortVignette != null)
72          {
73              comfortVignette.gameObject.SetActive(false);
74          }
75      }
76
77      private void ValidateControlInputs()
78      {
79          if (lever == null)
80          {
81              Debug.LogError($"Engine control lever not assigned on {
    gameObject.name}");
82              return;
83          }
84
85          if (knob == null)
```

```
86          {
87              Debug.LogError($"Direction control knob not assigned on {
    gameObject.name}");
88              return;
89          }
90
91          // Emergency stop is optional
92          if (emergencyStop != null)
93          {
94              Debug.Log($"Emergency stop configured for {gameObject.name}
    ");
95          }
96      }
97
98      private void SetupAudioSources()
99      {
100         if (AudioManager.instance != null && !string.IsNullOrEmpty(
    engineLoopSound))
101         {
102             // Pre-configure engine loop audio
103             engineAudioSource.clip = AudioManager.instance.sounds.
104                 FirstOrDefault(s => s.name == engineLoopSound)?.clip;
105         }
106     }
107
108     private void SetupInitialState()
109     {
110         // Initialize engine effects
111         if (mainEngineEffect != null)
112         {
113             mainEngineEffect.Stop();
114         }
115
116         foreach (var thruster in sideThrusters)
117         {
118             if (thruster != null)
119             {
120                 thruster.Stop();
121             }
122         }
123
124         // Initialize engine light
125         if (engineLight != null)
126         {
127             engineLight.enabled = false;
128         }
129
130         UpdateControlVisuals(false);
131     }
132
133     private void RegisterControlEvents()
134     {
135         if (emergencyStop != null)
136         {
137             emergencyStop.GetComponent<XRSimpleInteractable>().
    selectEntered.AddListener(
138                 args => ActivateEmergencyStop());
139         }
```

```
140     }
141
142     void Update()
143     {
144         UpdateEngineState();
145         UpdateMovementCalculations();
146         ApplyMovement();
147         UpdateAudioSystems();
148         UpdateVisualEffects();
149         UpdateComfortSettings();
150         HandleEnvironmentResponse();
151     }
152
153     private void UpdateEngineState()
154     {
155         if (emergencyStopActivated)
156         {
157             engineActive = false;
158             lever.value = false; // Force lever to off position
159         }
160         else
161         {
162             engineActive = lever.value;
163         }
164
165         // Detect engine state changes
166         if (engineActive != wasEngineActive)
167         {
168             HandleEngineStateChange(engineActive);
169             wasEngineActive = engineActive;
170         }
171     }
172
173     private void UpdateMovementCalculations()
174     {
175         targetVelocity = Vector3.zero;
176
177         if (engineActive)
178         {
179             // Calculate forward velocity based on engine state
180             float forwardVelocity = forwardSpeed * speedCurve.Evaluate
    (1f);
181
182             // Calculate side velocity based on knob position
183             currentKnobValue = knob.value; // knob.value typically
    ranges from 0 to 1
184             float normalizedKnob = Mathf.Lerp(-1f, 1f, currentKnobValue
    );
185             float sideVelocity = sideSpeed * normalizedKnob;
186
187             // Combine velocities
188             targetVelocity = new Vector3(sideVelocity, 0,
    forwardVelocity);
189
190             // Apply rotation if enabled
191             if (enableRotation)
192             {
193                 float rotationAmount = rotationSpeed * normalizedKnob *
```

```
    Time.deltaTime;
194                 transform.Rotate(Vector3.up, rotationAmount);
195             }
196 
197             // Clamp to maximum velocity
198             targetVelocity = Vector3.ClampMagnitude(targetVelocity,
    maxVelocity);
199         }
200     }
201 
202     private void ApplyMovement()
203     {
204         // Smooth velocity interpolation
205         float lerpSpeed = engineActive ? acceleration : deceleration;
206         currentVelocity = Vector3.Lerp(currentVelocity, targetVelocity,
    lerpSpeed * Time.deltaTime);
207 
208         // Apply movement to transform
209         Vector3 movement = currentVelocity * Time.deltaTime;
210         transform.position += transform.TransformDirection(movement);
211     }
212 
213     private void HandleEngineStateChange(bool newState)
214     {
215         if (newState)
216         {
217             StartEngine();
218         }
219         else
220         {
221             StopEngine();
222         }
223 
224         UpdateControlVisuals(newState);
225     }
226 
227     private void StartEngine()
228     {
229         // Play engine start sound
230         if (AudioManager.instance != null)
231         {
232             AudioManager.instance.Play(engineStartSound);
233 
234             // Start engine loop after brief delay
235             StartCoroutine(StartEngineLoopDelayed(0.5f));
236         }
237 
238         // Start engine visual effects
239         if (mainEngineEffect != null)
240         {
241             mainEngineEffect.Play();
242         }
243 
244         if (engineLight != null)
245         {
246             engineLight.enabled = true;
247             StartCoroutine(FadeEngineLight(0f, 1f, 0.3f));
248         }
```

```
249
250         // Enable comfort vignette if needed
251         if (enableComfortVignette)
252         {
253             ShowComfortVignette();
254         }
255     }
256
257     private void StopEngine()
258     {
259         // Stop engine audio
260         if (engineAudioSource != null)
261         {
262             engineAudioSource.Stop();
263         }
264
265         if (AudioManager.instance != null)
266         {
267             AudioManager.instance.Play(engineStopSound);
268         }
269
270         // Stop engine visual effects
271         if (mainEngineEffect != null)
272         {
273             mainEngineEffect.Stop();
274         }
275
276         foreach (var thruster in sideThrusters)
277         {
278             if (thruster != null)
279             {
280                 thruster.Stop();
281             }
282         }
283
284         if (engineLight != null)
285         {
286             StartCoroutine(FadeEngineLight(1f, 0f, 0.3f));
287         }
288
289         // Hide comfort vignette
290         if (enableComfortVignette)
291         {
292             HideComfortVignette();
293         }
294     }
295
296     private IEnumerator StartEngineLoopDelayed(float delay)
297     {
298         yield return new WaitForSeconds(delay);
299
300         if (engineActive && engineAudioSource != null)
301         {
302             engineAudioSource.Play();
303             yield return StartCoroutine(FadeAudioVolume(
    engineAudioSource, 0f, 0.8f, 0.5f));
304         }
305     }
```

```
306
307     private void UpdateAudioSystems()
308     {
309         if (engineAudioSource != null && engineActive)
310         {
311             // Modulate engine audio based on side movement
312             float pitchVariation = 1f + (Mathf.Abs(currentKnobValue -
    0.5f) * 0.2f);
313             engineAudioSource.pitch = pitchVariation;
314
315             // Adjust volume based on current velocity
316             float velocityRatio = currentVelocity.magnitude /
    maxVelocity;
317             engineAudioSource.volume = Mathf.Lerp(0.3f, 0.8f,
    velocityRatio);
318         }
319
320         // Handle side thruster audio
321         if (engineActive && Mathf.Abs(currentKnobValue - 0.5f) > 0.1f)
322         {
323             if (AudioManager.instance != null && !string.IsNullOrEmpty(
    thusterSound))
324             {
325                 // Play thruster sound with appropriate timing
326                 if (Time.frameCount % 30 == 0) // Play every 30 frames
    to avoid spam
327                 {
328                     AudioManager.instance.Play(thusterSound);
329                 }
330             }
331         }
332     }
333
334     private void UpdateVisualEffects()
335     {
336         if (!engineActive) return;
337
338         // Update side thruster effects based on knob position
339         float thrusterIntensity = Mathf.Abs(currentKnobValue - 0.5f) *
    2f; // 0 to 1 range
340
341         for (int i = 0; i < sideThrusters.Length; i++)
342         {
343             if (sideThrusters[i] != null)
344             {
345                 var emission = sideThrusters[i].emission;
346
347                 // Determine which thrusters should be active based on
    direction
348                 bool shouldBeActive = (i % 2 == 0) ?
349                     currentKnobValue < 0.4f : currentKnobValue > 0.6f;
350
351                 if (shouldBeActive)
352                 {
353                     if (!sideThrusters[i].isPlaying)
354                     {
355                         sideThrusters[i].Play();
356                     }
```

```
357                     emission.rateOverTime = thrusterIntensity * 50f;
358                 }
359                 else
360                 {
361                     sideThrusters[i].Stop();
362                 }
363             }
364         }
365 
366         // Update main engine effect intensity
367         if (mainEngineEffect != null)
368         {
369             var emission = mainEngineEffect.emission;
370             float engineIntensity = currentVelocity.magnitude / maxVelocity;
371             emission.rateOverTime = Mathf.Lerp(20f, 100f, engineIntensity);
372         }
373     }
374 
375     private void UpdateComfortSettings()
376     {
377         if (!enableComfortVignette || comfortVignette == null) return;
378 
379         // Adjust vignette intensity based on velocity and direction changes
380         float velocityRatio = currentVelocity.magnitude / maxVelocity;
381         float directionChange = Mathf.Abs(currentKnobValue - 0.5f) * 2f;
382         float targetIntensity = Mathf.Max(velocityRatio, directionChange) * vignetteIntensity;
383 
384         // Apply vignette intensity (this would require a custom vignette shader or post-processing)
385         ApplyVignetteIntensity(targetIntensity);
386     }
387 
388     private void HandleEnvironmentResponse()
389     {
390         if (!enableEnvironmentParallax || asteroidField == null) return;
391 
392         // Move asteroid field in opposite direction to create parallax effect
393         Vector3 environmentMovement = -currentVelocity * fieldMovementMultiplier * Time.deltaTime;
394         asteroidField.position += environmentMovement;
395 
396         // Wrap asteroid field position to prevent it from moving too far
397         WrapAsteroidFieldPosition();
398     }
399 
400     private void ShowComfortVignette()
401     {
402         if (comfortVignette != null && vignetteCoroutine == null)
403         {
404             vignetteCoroutine = StartCoroutine(FadeVignette(0f,
```

```
    vignetteIntensity, vignetteBlendTime));
405         }
406     }
407 
408     private void HideComfortVignette()
409     {
410         if (comfortVignette != null && vignetteCoroutine == null)
411         {
412             vignetteCoroutine = StartCoroutine(FadeVignette(
    vignetteIntensity, 0f, vignetteBlendTime));
413         }
414     }
415 
416     private void UpdateControlVisuals(bool active)
417     {
418         // Update control panel materials
419         Renderer leverRenderer = lever.GetComponent<Renderer>();
420         if (leverRenderer != null)
421         {
422             leverRenderer.material = active ? activeMaterial :
    inactiveMaterial;
423         }
424     }
425 
426     public void ActivateEmergencyStop()
427     {
428         emergencyStopActivated = true;
429 
430         if (AudioManager.instance != null)
431         {
432             AudioManager.instance.Play("EmergencyStop");
433         }
434 
435         Debug.Log("Emergency stop activated!");
436 
437         // Auto-reset emergency stop after delay
438         StartCoroutine(ResetEmergencyStop(3f));
439     }
440 
441     private IEnumerator ResetEmergencyStop(float delay)
442     {
443         yield return new WaitForSeconds(delay);
444         emergencyStopActivated = false;
445         Debug.Log("Emergency stop reset");
446     }
447 
448     // Utility Coroutines
449     private IEnumerator FadeEngineLight(float fromIntensity, float
    toIntensity, float duration)
450     {
451         float timer = 0f;
452         while (timer < duration)
453         {
454             timer += Time.deltaTime;
455             float progress = timer / duration;
456             engineLight.intensity = Mathf.Lerp(fromIntensity,
    toIntensity, progress);
457             yield return null;
```

```
458         }
459         engineLight.intensity = toIntensity;
460 
461         if (toIntensity == 0f)
462         {
463             engineLight.enabled = false;
464         }
465     }
466 
467     private IEnumerator FadeAudioVolume(AudioSource source, float
    fromVolume, float toVolume, float duration)
468     {
469         float timer = 0f;
470         while (timer < duration)
471         {
472             timer += Time.deltaTime;
473             float progress = timer / duration;
474             source.volume = Mathf.Lerp(fromVolume, toVolume, progress);
475             yield return null;
476         }
477         source.volume = toVolume;
478     }
479 
480     private IEnumerator FadeVignette(float fromIntensity, float
    toIntensity, float duration)
481     {
482         if (comfortVignette != null)
483         {
484             comfortVignette.gameObject.SetActive(true);
485         }
486 
487         float timer = 0f;
488         while (timer < duration)
489         {
490             timer += Time.deltaTime;
491             float progress = timer / duration;
492             float currentIntensity = Mathf.Lerp(fromIntensity,
    toIntensity, progress);
493             ApplyVignetteIntensity(currentIntensity);
494             yield return null;
495         }
496 
497         ApplyVignetteIntensity(toIntensity);
498 
499         if (toIntensity == 0f && comfortVignette != null)
500         {
501             comfortVignette.gameObject.SetActive(false);
502         }
503 
504         vignetteCoroutine = null;
505     }
506 
507     private void ApplyVignetteIntensity(float intensity)
508     {
509         // Implementation would depend on vignette solution
510         // Could use post-processing volume, custom shader, or UI
    overlay
511         if (comfortVignette != null)
```

```
        {
            var canvasGroup = comfortVignette.GetComponent<CanvasGroup
    >();
            if (canvasGroup != null)
            {
                canvasGroup.alpha = intensity;
            }
        }
    }

    private void WrapAsteroidFieldPosition()
    {
        float wrapDistance = 100f;
        Vector3 pos = asteroidField.position;

        if (pos.magnitude > wrapDistance)
        {
            asteroidField.position = pos.normalized * (wrapDistance *
    0.1f);
        }
    }

    void OnDisable()
    {
        // Cleanup when component is disabled
        if (engineAudioSource != null)
        {
            engineAudioSource.Stop();
        }

        if (vignetteCoroutine != null)
        {
            StopCoroutine(vignetteCoroutine);
            vignetteCoroutine = null;
        }
    }
}
```

Listing 10: Sophisticated VR Vehicle Navigation System

Sophisticated Multi-Input Control System:

The spaceship controller demonstrates advanced VR interaction design through its combination of lever and knob controls that provide intuitive vehicle operation. The lever provides binary engine control (on/off) while the knob enables analog directional control, creating a realistic spacecraft interface that maps naturally to user expectations while remaining comfortable in VR.

Comprehensive Comfort Features:

The implementation includes multiple comfort features specifically designed for VR usage. The comfort vignette system reduces peripheral vision during movement, helping to prevent motion sickness. The graduated acceleration and deceleration curves provide smooth movement transitions that feel natural while avoiding sudden changes that could cause discomfort.

Advanced Audio Design:

The audio system demonstrates sophisticated understanding of VR audio requirements through its multi-layered approach. The engine loop audio includes pitch and

volume modulation based on movement state, while directional thrusters provide audio cues for control input. The spatialized audio integration maintains immersion while providing functional feedback about ship systems.

Environmental Integration:

The parallax asteroid field system creates the illusion of movement through space without requiring physically large environments. This technique is particularly valuable in VR where physical space constraints limit traditional level design approaches. The wrapping system prevents the environment from becoming disconnected from the player while maintaining the illusion of continuous movement.

4.8 Game Over Condition and Hint

If the player fails to escape before the timer reaches zero, a Game Over screen appears. This screen displays a helpful hint to encourage replay and improvement.

> *Remember to check for hidden oxygen supplies and solve all puzzles quickly to maximize your survival time!*

A short video demonstrating this Game Over screen and hint can be viewed at:

Watch the Game Over example here

5 Conclusion

5.1 Project Summary

"Space Survival" represents a comprehensive implementation of advanced virtual reality development principles that successfully balances technical sophistication with user experience requirements. Through systematic attention to performance optimization, user comfort considerations, and engaging interaction design, the project demonstrates mastery of VR development fundamentals while showcasing innovative approaches to common VR challenges.

The project's architecture demonstrates professional-level understanding of software engineering principles through its modular component design, efficient system integration, and comprehensive error handling. The extensive code documentation and detailed implementation analysis provide valuable insights into the decision-making processes that drive successful VR development projects.

5.2 Technical Innovation

The innovative aspects of Space Survival extend beyond conventional VR development through several key contributions:

The custom XR socket interaction system provides enhanced control over object placement while maintaining compatibility with Unity's XR Interaction Toolkit. This approach demonstrates how to extend existing frameworks rather than replacing them, providing a sustainable development approach that benefits from ongoing framework improvements while addressing specific project requirements.

The multi-modal feedback systems throughout the project showcase advanced understanding of VR user experience principles. By providing visual, audio, and haptic

feedback for all major interactions, the implementation ensures accessibility across different user preferences while creating rich, immersive experiences that maintain presence.

The performance optimization strategies demonstrate deep understanding of VR technical requirements through systematic attention to frame rate stability, memory management, and resource optimization. These implementations provide valuable patterns that can be applied to future VR development projects.

6 Demo Access

A video demonstration of Space Survival is available here (done with Meta XR Simulator):

Watch the Space Survival Demo

7 References

- Unity Documentation: `https://docs.unity3d.com/`
- Meta XR SDK Docs: `https://developer.oculus.com/documentation/unity/unity-overview/`
- XR Interaction Toolkit: `https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.5/manual/index.html`
- Meta XR Simulator : `https://assetstore.unity.com/packages/tools/integration/meta-xr-simulator-266732`
- Valem Let's Make a VR Game : `https://www.youtube.com/watch?v=QCvqimfrMZw&list=PLpEoiloH-4eM-fykn_3_QcJ-A_MIJF5B9&index=1`
- Sci-Fi Styled Modular Pack : `https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-styled-modular-pack-82913`
- VR-Game-Jam-Template : `https://github.com/ValemVR/VR-Game-Jam-Template`