Text Script for “Virtual Experience Engine”

Table of Contents

Topic Number
Title

Introduction: What is a Virtual Experience Engine?	1
ThreeJS + CannonJS Overview	2
Engine Comparison and Advantages	3
Virtual Tours: 360 Panoramic Image Integration	4
Model Viewer: 3D Assets Integration	5
Simulated Interiors: Cubemap Parallax Integration	6
Virtual Experience: Virtual Experience Level Integration	7
Virtual Experience User Interface (UI)	8
Introduction to Drag and Drop Level Prototyping	9
Drag and Drop Avatar System	10
Vehicle Drag and Drop into the Virtual Experience Engine	11
Advantages of the Virtual Experience Engine	12
Modular Pipeline with Asynchronous Physics	13
Asynchronous Physics in Detail	14
Asset Optimization and Mesh Management	15
Performance Optimization Techniques	16
Deployment and Browser Compatibility	17

Case Studies and Example Projects 18

UI/UX for Interactive Virtual Environments 19

Integrating External Data (BIM, GIS, IoT) 20

Future Trends in Virtual Experience Engines 21

Title

ThreeJS + CannonJS
Virtual Experience Engine Script

1: Introduction:

What is a Virtual Experience Engine?

This class is intended for architectural designers, developers, and engineers who are either new to Virtual Experience Engines (VEEs) or require a structured refresher on virtual experience architecture and workflows. It provides a hands-on, practical approach to understanding how modern 3D engines operate, with a particular focus on **interactive visualization, immersive project presentation, and browser-based accessibility**.

A **Virtual Experience Engine** is a system or framework that allows designers and engineers to bring 3D models and spatial designs to life in an **interactive, dynamic, and explorable** manner. Unlike static renders or pre-recorded walkthroughs, VEEs support real-time interaction with 3D content, enabling users to navigate environments, manipulate objects, test spatial scenarios, and experience designs in a way that more closely simulates real-world conditions.

The primary focus of this course is **browser-based showcasing of architectural and design projects**. By the end of this class, participants will be able to create environments that extend far beyond static visualization, producing immersive, explorable 3D spaces that can be accessed across devices without specialized software installations.

Application Areas

Virtual Experience Engines are applicable across a wide range of design disciplines:

BIM Models
VEEs enable detailed, interactive exploration of building information models. Designers can inspect structural elements, room layouts, material finishes, and design constraints within a navigable environment. Stakeholders can explore spatial relationships between systems, simulate maintenance access, and understand construction logic before physical execution.

Urban Designs
Urban planners and architects can explore entire districts or cities in real time. VEEs support simulation of pedestrian movement, traffic flow, zoning constraints, daylight exposure, and density studies. Designers can test how sunlight interacts with building heights throughout the day or evaluate accessibility in complex urban conditions.

Interiors and Exteriors
From single-room interiors to multi-level structures, VEEs allow designers to test furniture layouts, materials, lighting conditions, and spatial ergonomics interactively.

Users can walk through spaces, adjust lighting in real time, and experience scale and circulation as if physically present.

Event Spaces
Event planners can design and test layouts for conferences, exhibitions, and public gatherings. VEEs support simulations of crowd flow, visibility, booth placement, and emergency exit routes, enabling safer and more efficient spatial planning.

Product Presentations
Designers and stakeholders can interact with product prototypes virtually, inspecting scale, materials, colors, and mechanical behavior before physical production. This reduces development costs and accelerates feedback cycles.

Engine Foundation and Transferable Skills

The course uses a **web-based Virtual Experience Engine built with ThreeJS and CannonJS** as a practical reference implementation. The engine is modular, lightweight, and browser-accessible, allowing participants to experiment with real-time environments while receiving immediate visual feedback.

Skills developed in this course are **fully transferable** to other engines such as Unreal Engine, Unity, Godot, or any platform that supports interactive 3D systems. The emphasis is on **concepts, workflows, and architectural understanding**, rather than engine-specific limitations.

Recommended Background

Participants are recommended to have:

- General computer skills and familiarity with web browsers
- Basic knowledge of 3D modeling tools such as 3ds Max, SketchUp, Blender, Rhino, or equivalent software
- Optional coding knowledge for extending interaction or logic (helpful but not required)
- A background in architecture, spatial design, or 3D visualization is advantageous

Learning Outcomes

By the end of this chapter, participants will understand:

- What a Virtual Experience Engine is and how it differs from static visualization
- How VEEs are applied in architectural, urban, interior, event, and product workflows
- The core skills and tools required to operate and experiment with a VEE
- The advantages of browser-based interactive visualization, including rapid prototyping, real-time feedback, and cross-platform access

Example Use Case

Imagine an architectural firm preparing a pitch for a new multi-use building. Using a Virtual Experience Engine, the team can:

1. Load the BIM model into a browser-based environment
2. Walk clients through every floor and room interactively
3. Adjust materials, lighting, or furniture layouts in real time
4. Simulate crowd flow for an event space
5. Export an interactive version that clients can explore independently

By leveraging a VEE, the design becomes a **living, interactive representation**, offering a far more compelling experience than drawings, renders, or videos.

2: ThreeJS + CannonJS Overview

Introduction

In this chapter, we examine the core technologies that form the backbone of the Virtual Experience Engine: **ThreeJS for rendering** and **CannonJS for physics simulation**. These technologies define how interactive 3D content is displayed, navigated, and physically behaves within a browser environment.

While the examples use ThreeJS and CannonJS, the workflows and architectural patterns discussed here are applicable to other engines such as Unity, Unreal Engine, or Godot.

2.1 ThreeJS: Browser-Based 3D Rendering

ThreeJS is a JavaScript library that enables interactive 3D graphics in web browsers using WebGL or WebGPU. Its modular design and extensive ecosystem make it a popular choice for architectural walkthroughs, real-time visualization, and rapid 3D prototyping.

Key Features

- Scene graph management for hierarchical object organization
- Mesh rendering with multiple geometries and materials
- Perspective and orthographic camera systems
- Lighting systems including ambient, directional, point, and spotlights
- Physically Based Rendering (PBR) materials and texture support
- Animation systems for keyframe, skeletal, and morph targets
- Post-processing effects such as bloom and ambient occlusion

Example Use Case
An interior designer loads furniture models, applies realistic materials, adds daylight simulation, and enables camera navigation so clients can experience the space interactively.

2.2 CannonJS: Physics Simulation

CannonJS is a lightweight JavaScript physics engine designed for real-time rigid body simulation in browser environments.

Key Features

- Rigid body dynamics with mass and inertia
- Collision detection using simplified geometry
- Constraints such as hinges and sliders
- Gravity, impulses, and force application
- Optional Web Worker support for performance optimization

Example Use Case
An architect tests a movable partition system in an event hall, observing how it reacts to collisions and constraints to validate safety and usability.

2.3 Integration: ThreeJS + CannonJS

The Virtual Experience Engine separates **visual rendering** from **physical simulation**, allowing both systems to operate efficiently and independently.

Typical Workflow

1. Load GLTF meshes into the scene
2. Assign physics bodies to designated collision meshes
3. Update physics simulation per frame
4. Synchronize mesh transforms with physics bodies
5. Render the updated scene

Key Benefits

- Smooth real-time interaction
- Accurate collision and gravity simulation
- Modular system architecture

2.4 Performance Optimizations

To maintain performance in browser-based environments:

- Mesh simplification and polygon reduction
- Level of Detail (LOD) switching
- Texture mipmapping and resolution management
- Offloading physics calculations to Web Workers

Example optimization trigger during user interaction:

```
if (event.type.match(/(mousedown|touchstart)/)) {
  setTimeout(() => {
    _pm.model.optimizeAssets({});
  }, 1024);
}
```

2.5 Hands-On Exercise

Goal: Create a simple scene with a bouncing ball.

Steps:

1. Create a ThreeJS scene, camera, and renderer
2. Add a floor plane
3. Add a sphere mesh with a CannonJS rigid body
4. Apply gravity and collision detection
5. Animate and synchronize physics

Expected Outcome:
The ball falls, collides, and bounces realistically, demonstrating core VEE principles.

22: Building Virtual Worlds with GLTF Assets in a Browser-Based Virtual Experience Engine

This section extends the engine fundamentals by focusing on **GLTF-based world construction** within the browser-based Virtual Experience Engine.

GLTF Structure in the Engine

In the demo environment, a GLTF model represents more than visible geometry. Each level is composed of multiple functional components:

Physics Bodies (phy-bodies)
Simplified meshes used by CannonJS as colliders, defining walkable surfaces and physical boundaries.

Plane Axes
Used for avatar spawn points, grounding, orientation, and spatial reference.

Visible Meshes
Rendered geometry optimized for real-time performance through baking and texture optimization.

This separation enables contributors to understand how **structure, physics, and visuals** work together inside a ThreeJS-based virtual environment.

Supported Design Tools

Any 3D modeling or CAD software may be used, including:

- 3ds Max
- SketchUp
- Rhino
- Maya
- Blender
- AutoCAD

Models are exported to GLTF (commonly via Blender) and used directly in the engine.

Drag-and-Drop Workflow

Participants can:

- Use GLTF assets from Sketchfab
- Export models from their preferred tools
- Drag and drop GLTF files directly into the browser

All processing occurs **locally**. No files are uploaded or stored externally, making the environment suitable for private experimentation and learning.

Purpose of the Environment

Because the environment is intentionally minimal, contributors can focus on:

- Testing scale and spatial layout
- Validating real-time performance
- Understanding GLTF behavior in a ThreeJS workflow

3: Engine Comparison and Advantages

In the next chapter, the Virtual Experience Engine is compared to traditional engines such as Unity, Unreal Engine, and Godot. While those platforms offer extensive features, the VEE emphasizes **modularity, browser accessibility, and rapid iteration**, making it especially suitable for architectural and design-driven workflows.

4: Virtual Tours:
360 Panoramic Image Integration

This chapter focuses on the creation of a virtual tour through the integration of 360-degree panoramic imagery into the virtual experience engine. Such integration enables effective project walkthrough presentations and the development of immersive game environments.

```
_loader = (d) => {
    // Initial settings here...
    return {
        pIndex: d.index,
        program: [
            {
                environment: {
                    // Environment settings
                },
                camera: {
                    //  Camera settings here...
                },
                tIndex: 0, // Tours index
                tours: [
                    {
                        name: _pm.mobile ?
                            // 4k for mobile
                            "garage_360_4k_000.jpg" :
                            // 8k for desktop
                            "garage_360_8k_000.jpg",
                        environment: {
                            name: "gray_wide_street_01_256.hdr",
                            // HDR orientation
                            rotation: 0
                        },
                        // Marker or pin position
                        position: { x: 0, y: 1.55, z: 3.5 },
                        // Panorama orientation
                        rotation: 90
                    }
                ],
                markers: [
                    // Pins
                    {
                        image: {
                            path: "assets/textures/sprites/",
                            name: "spining-circle.gif",
                            size: { length: .64, height: .64 },
                            visible: !0
                        },
                        details: {
                            title: "Starting Point"
                        },
                        position: { x: 0, y: 0, z: 3.5 },
                        rotation: { x: 0, y: 0, z: 0 },
                    }
                ],
                operations: {
                    "markers-starting-point": {
                        // Type of operation
                        name: "tour-360",
                        // Target index
                        tours: 0,
                        hide: [
                            // Hide itself
                            "markers-starting-point",
```

```
                        // Hide others...
                        "markers-other-1",
                        "markers-other-2",
                    ],
                }
            }
        }
    ]
}
}
```

For the engine to locate the 360-degree panoramic images, they must be stored in the **`assets\textures\equirectangular`** directory. Multiple panoramic images and interactive markers are required to allow navigation between scenes, creating the illusion of a virtual tour.

5: Model Viewer:
3D Assets Integration

In this chapter, we will create a browser-based model viewer that allows users to view and interact with 3D models directly within a web browser. This viewer is useful for presenting 3D assets in a clear and accessible way, making it ideal for demonstrations, design reviews, and interactive presentations.

```
_loader = (d) => {
    // Initial settings here...
    return {
        pIndex: d.index,
        program: [
            {
                environment: {
                    // Environment settings
                },
                camera: {
                    //  Camera settings here...
                },
                tIndex: 0, // Tours index
                mIndex: {
                    selected: 0,
                },
                models: {
                    0: {
                        // Asset name
                        name: "pile_of_old_tires",
                        preloaded: !0,
                        // Position in meters Y-UP
                        position: { x: 0, y: 0, z: 0 },
                        // Rotation in degress Y-UP
                        rotation: { x: 0, y: 180, z: 0 },
                        // Level of details is 24%
                        lod: .21
                    },
                }
            }
```

```
        ]
    }
}
```

The 3D assets must be placed in the **`assets\models`** folder for the engine to detect them. Ensure that each model is in GLTF binary format with the `.glb` extension, as the engine is designed to support `.glb` files only. If your 3D assets are in other formats such as `.3ds` or `.skp`, you must first export them as FBX files, then import them into Blender and re-export them as `.glb` *(GLTF binary format)*.

6: Simulated Interiors:
Cubemap Parallax Integration

In this chapter, we will add a simulated interior parallax effect that creates the illusion of depth inside interior spaces. This technique enhances visual realism by making interior elements appear to shift relative to the viewer's perspective, even though the geometry is simplified.

```
_loader = (d) => {
    // Initial settings here...
    return {
        pIndex: d.index,
        program: [
            {
                environment: {
                    // Environment settings
                },
                camera: {
                    //  Camera settings here...
                },
                tIndex: 0, // Tours index
                mIndex: {
                    selected: 0,
                },
                models: {
                    0: {
                        name: "large screen",
                        preloaded: !0,
                        position: { x: 0, y: 0, z: -3 },
                        rotation: { x: 0, y: 0, z: 0 },
                        interiors: [
                            {
                                name: "large-screen",
                                cube: ["2.jpg", "1.jpg", "4.jpg", "3.jpg", "6.jpg", "5.jpg"],
                                path: "./assets/textures/cube/interiors/lobby 1/",
                                flipY: !0,
                                fov: 1
                            },
                        ],
                    }
                }
            }
        ]
    }
}
```

We will implement an interior parallax effect using a simple plane or a GLTF plane model. The plane serves as the surface onto which the interior parallax shader is applied, creating the illusion of depth inside the space without fully modeling the interior geometry. The interior parallax textures must be placed in the **`assets/textures/cube/interiors/lobby1`** directory, where the engine will load them automatically. This technique allows flat surfaces to appear as detailed interior environments while maintaining high performance, making it ideal for real-time rendering and virtual experiences.

7: Virtual Experience:
Virtual Experience Level Integration

In this chapter, we will load a 3D model that acts as the map or level of the virtual environment. This model represents the space where the avatar can move and interact with elements such as walls and floors. To handle movement and collisions realistically, we will use Cannon.js, a physics library that allows the avatar to walk on floors and stop at walls instead of passing through them.

The 3D asset must be saved in .glb format. In addition to the detailed visual model, it should include simplified versions of the walls, floors, and other structural elements. These simplified meshes are used only for collision and interaction, ensuring smooth performance while preserving the visual quality of the design. This approach is commonly used in real-time visualization to balance accuracy and efficiency.

```
_loader = (d) => {
    // Initial settings here...
    return {
        pIndex: d.index,
        program: [
            {
                environment: {
                    // Environment settings
                },
                camera: {
                    //  Camera settings here...
                },
                tIndex: 0, // Tours index
                mIndex: {
                    selected: 0,
                },
                models: {
                    0: {
                        // Asset or level name
                        name: "apartment",
                        preloaded: !0,
                        userData: {
                            // Sets the wall as obstacle for camera
                            isObstacle: !0,
                            // Sets the assets as navigation model
                            // by clicking the mouse wheel will show
                            // navigation icon.
                            isNavigation: !0
                        },
                        videos: [
                            {
```

```
                            // Target material name for
                            // video texture
                            name: "television",
                            // Video file source
                            url: `assets/textures/videos/closer you & i.mp4`,
                            // Adjust video intensity
                            emissiveIntensity: .64,
                            flipX: !1,
                            flipY: !1
                        }
                    ],
                    position: { x: 0, y: 0, z: 0 },
                    rotation: { x: 0, y: 0, z: 0 },
                    // scale: { x: 1, y: 1, z: 1 },
                    // Level of details 96%
                    lod: .96,
                    // Enable physics, allow gltf model
                    // colliders to be added in cannon.js
                    physics: !0
                }
            }
        }
    ]
}
}
```

The 3D assets must be placed in the **`assets\models`** folder in GLTF binary format (`.glb`), as the engine only supports .glb files. If your original model is in another format such as .3ds or .skp, first export it as FBX, import it into Blender, and then export it as `.glb`. Each model should include a simplified collision mesh for walls, floors, and other solid surfaces, which must be named exactly **`phy-body-main-trimesh-0kg-wall-0001`** *(explore apartment.glb)* so the engine can automatically apply physics using *Cannon.js*. Any material intended to display video must be assigned in Blender and its name referenced in the engine settings *(refer to the above code)*, allowing the engine to replace that material with dynamic video content at runtime.

8: Virtual Experience User Interface (UI):
8.1 Introduction

A Virtual Experience Engine is not complete without a clear and intuitive **User Interface (UI)**. While rendering and physics define how a virtual world looks and behaves, the UI defines **how users interact with that world**. In browser-based VEEs, the UI plays a critical role in accessibility, usability, and onboarding, especially for non-technical users such as architects, clients, and stakeholders.

This chapter focuses on the design and implementation of a **minimal yet functional UI layer** that supports navigation, interaction, and user guidance within a browser-based virtual experience. The UI is designed to be lightweight, non-intrusive, and adaptable across desktop and mobile devices.

Virtual Experience Engine URL:

https://theneoverse.web.app/#threeviewer&&construct

8.2 Input Systems Overview

The Virtual Experience Engine supports multiple input methods to ensure cross-platform accessibility and ease of use:

- Keyboard controls for desktop navigation
- Touch controls for mobile and tablet devices
- On-screen buttons for essential actions
- Instructional UI elements such as modals and overlays

By supporting multiple interaction modes, the engine allows users to explore environments regardless of device or technical background.

8.3 Keyboard Controls

Keyboard input provides precise and familiar navigation for desktop users, particularly architects and designers working with large-scale environments.

Figure 8.1 – Desktop UI

Common keyboard mappings include:

- **W / A / S / D**: Forward, left, backward, right movement

- **Arrow Keys**: Alternative directional movement
- **Q / E**: Side movement
- **Mouse wheel**: Roll switch camera, press to navigate
- **Spacebar**: Jump or brake
- **Shift**: Sprint or accelerate
- **[/]**: Look up or down
- **Enter**: Enter vehicle
- **Esc**: Exit vehicle
- **Z**: Wave or horn
- **L**: Location url
- **F / O**: Follow avater or vehicle, orbit freely
- **H**: Avatar studio
- **G**: Select avatar

Keyboard controls are typically paired with mouse input for camera orientation, enabling first-person or free-walk navigation through the virtual environment.

Design Consideration:
Keyboard controls should remain configurable and clearly documented to avoid confusion for first-time users.

8.4 Touch Controls (Mobile and Tablet)

Touch controls are essential for browser-based VEEs accessed on smartphones and tablets. These controls translate spatial navigation into intuitive gestures:

- **Virtual joystick or directional pad** for movement
- **Swipe gestures** for camera rotation
- **Tap interactions** for selection or UI activation

Touch input is designed to prioritize simplicity and minimize screen clutter. UI elements are positioned to avoid obstructing the scene while maintaining accessibility.

Use Case Example:
A client reviewing an architectural walkthrough on a tablet can navigate rooms using touch controls without requiring external peripherals.

8.5 On-Screen Buttons and Action Controls

On-screen buttons provide quick access to core actions and features without requiring keyboard input. These may include:

- Reset position or respawn
- Toggle camera modes (walk, orbit, free view)
- Enable or disable physics interactions
- Open help or instruction panels

Figure 8.2 – Mobile UI

Buttons are typically implemented as HTML or CSS overlays layered above the WebGL canvas, ensuring flexibility and responsiveness across screen sizes.

UI Principle:
Buttons should remain minimal and contextual, appearing only when necessary to avoid overwhelming the user.

8.6 Keyboard Instruction and Help Modals

To reduce onboarding friction, the Virtual Experience Engine includes **instruction modals** that explain controls and interaction logic.

Instruction modals may include:

- Keyboard and mouse mappings
- Touch gesture explanations
- Navigation tips and usage hints
- Context-specific instructions (e.g., exploration vs interaction modes)

These modals are usually displayed:

- On first load
- When the user presses a help button or shortcut
- When switching interaction modes

Instruction panels can be dismissed easily and reopened at any time, ensuring users remain in control of their experience.

8.7 UI Architecture and Implementation

The UI layer is intentionally decoupled from the rendering and physics systems. This separation allows:

- Independent UI updates without affecting frame rate
- Rapid iteration on layout and interaction design
- Easy customization for different projects or clients

Typical UI elements are implemented using:

- HTML for structure
- CSS for styling and responsiveness
- JavaScript for event handling and state control

This approach aligns with web development best practices and allows designers and developers to modify UI behavior without deep changes to the engine core.

8.8 Accessibility and Usability Considerations

When designing UI for a Virtual Experience Engine, accessibility is a primary concern. Best practices include:

- Clear visual contrast for buttons and text
- Large touch targets for mobile devices
- Consistent control mappings across scenes
- Non-blocking UI elements that preserve immersion

By prioritizing usability, the engine ensures that technical complexity does not become a barrier to spatial exploration and design evaluation.

8.9 Role of UI in Design Review and Presentation

The UI is not merely a control mechanism; it is a **presentation layer** that shapes how users perceive and engage with the virtual environment.

In professional contexts, the UI enables:

- Guided walkthroughs during client presentations
- Self-exploration without verbal instruction
- Rapid feedback during design reviews
- Clear separation between navigation and observation

A well-designed UI enhances confidence, reduces confusion, and allows stakeholders to focus on **design intent rather than controls**.

8.10 Summary

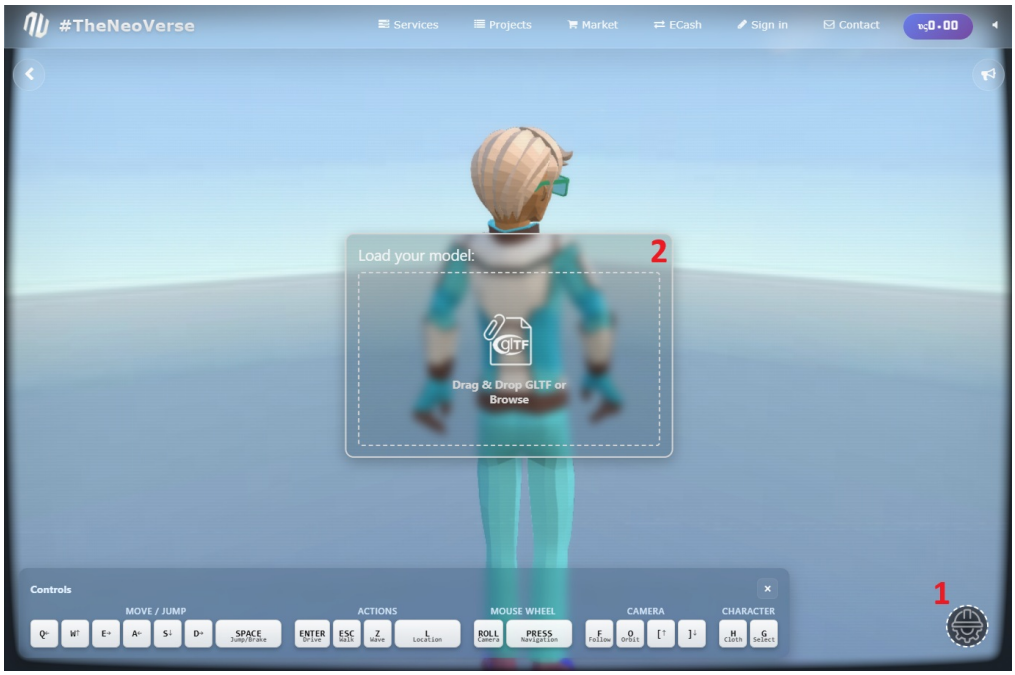
- The Virtual Experience UI enables navigation, interaction, and user guidance
- Keyboard, touch, and button-based controls ensure cross-platform accessibility
- Instruction modals reduce onboarding friction for non-technical users
- UI systems are modular, lightweight, and decoupled from rendering and physics
- Effective UI design enhances immersion, usability, and professional presentation

9. Introduction to Drag and Drop Level Prototyping

9.1 Introduction

Drag and drop loading is a core feature of the Virtual Experience Engine. It is designed to simplify the process of testing and validating 3D assets inside a real time browser environment. This workflow allows designers, architects, and developers to load prototype levels quickly without configuring build pipelines, asset managers, or complex scripts.

The purpose of this chapter is to introduce a simple prototype level that demonstrates how a GLTF asset can function as a complete virtual environment when it is dragged directly into the engine.



Figuere: 9.1 – The construct UI: construct button (1) shows drag & drop modal (2).

Rather than focusing on visual complexity, this chapter emphasizes functional clarity. The goal is to understand how geometry, physics, and spatial reference elements work together when a level is dynamically loaded.

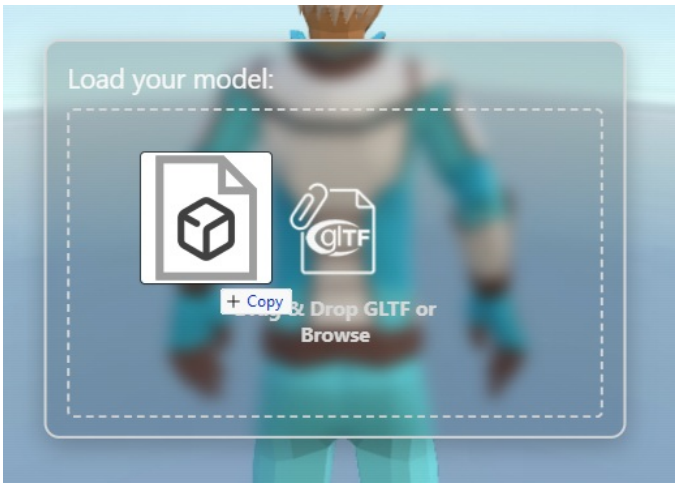


Figure 9.2 – Drag and dropping GLTF

9.2 Concept of a Drag and Drop Level

In the Virtual Experience Engine, a level is treated as a self contained GLTF package. When a GLTF file is dragged into the browser window, the engine performs the following actions:

1. The GLTF file is parsed and loaded in real time
2. Visual meshes are rendered in the scene
3. Physics colliders are created and registered
4. The avatar position is resolved
5. The environment becomes immediately explorable

This approach enables rapid iteration and allows contributors to test scale, orientation, and navigation behavior without restarting or recompiling the application.

9.3 Sample Prototype Level Structure

The prototype level used in this chapter is intentionally minimal. It consists of three essential parts that together define a functional virtual level:

1. A visible mesh that represents what the user sees
2. A simplified mesh that acts as the physics collider
3. A plane axis named avatar-here that defines avatar placement

Each of these parts has a distinct role inside the engine.

9.4 Visible Mesh

The visible mesh represents the display geometry of the level. This is the geometry rendered on screen and experienced visually by the user.

Typical visible meshes may include architectural elements, terrain, floors, walls, and props. These meshes may use textures, baked lighting, or optimized materials to achieve good visual quality while remaining suitable for real time rendering in the browser.

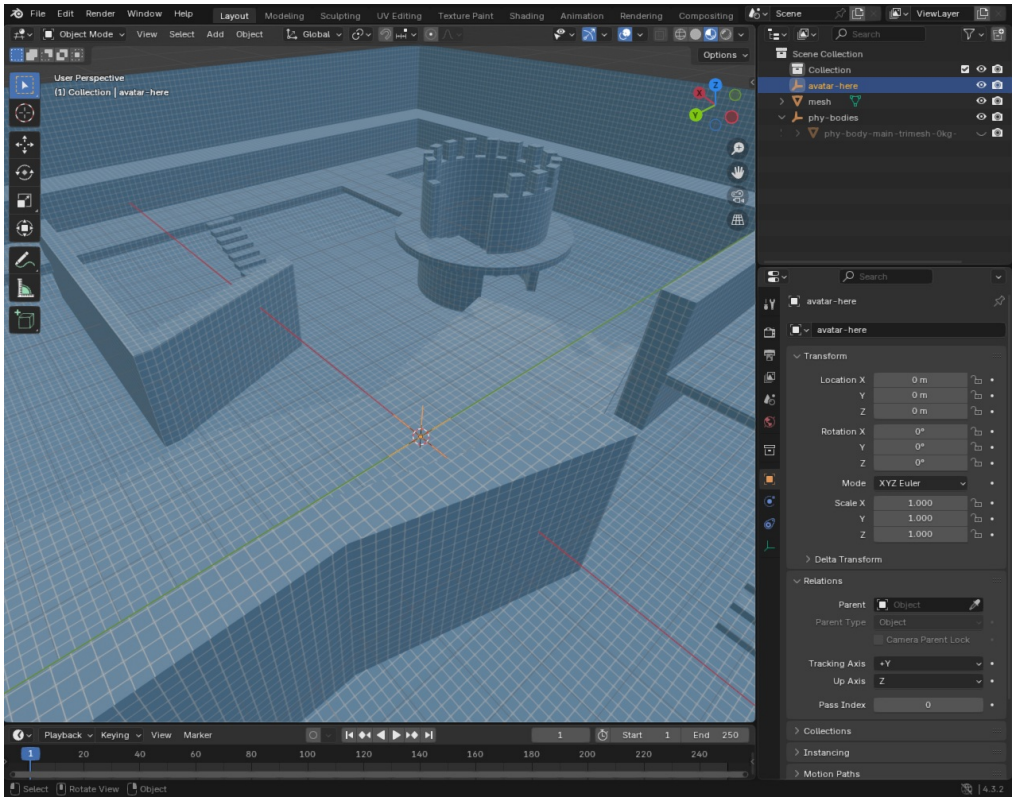


Figure 9.3 – Level Prototype “**mesh**”

Visible meshes are designed primarily for appearance and spatial understanding, not for physical accuracy.

9.5 Collider Mesh

The collider mesh is a simplified version of the visible mesh. Its role is to define how the avatar and other objects physically interact with the environment.

Characteristics of collider meshes include reduced polygon count, simplified topology, and stable shapes that are suitable for collision detection. These meshes are not rendered visually and exist only for physics calculations.

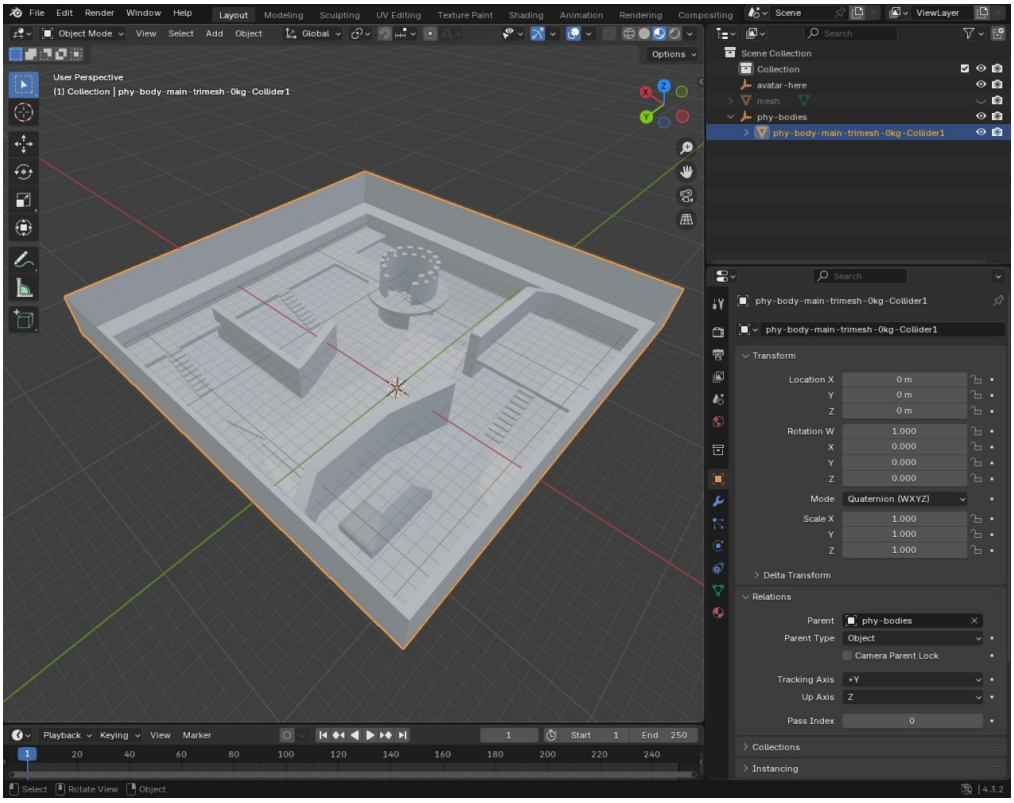


Figure 9.4 – Collider mesh "**`phy-body-main-trimesh-0kg-Collider1`**"

By separating collider geometry from visual geometry, the engine maintains better performance and more reliable physical behavior.

9.6 Plane Axis Named avatar-here

The third and most important element in the prototype level is a plane axis named avatar-here.

This plane defines the initial position where the avatar will be placed when the level is loaded. When a GLTF file is dragged and dropped into the engine, the system searches for an object with this exact name.

Once detected, the avatar is repositioned to the location of the avatar-here plane. This ensures that navigation always begins from a known and intentional position within the level.

Using a named plane for avatar placement removes the need for manual camera adjustment and guarantees consistent behavior across different prototype levels.

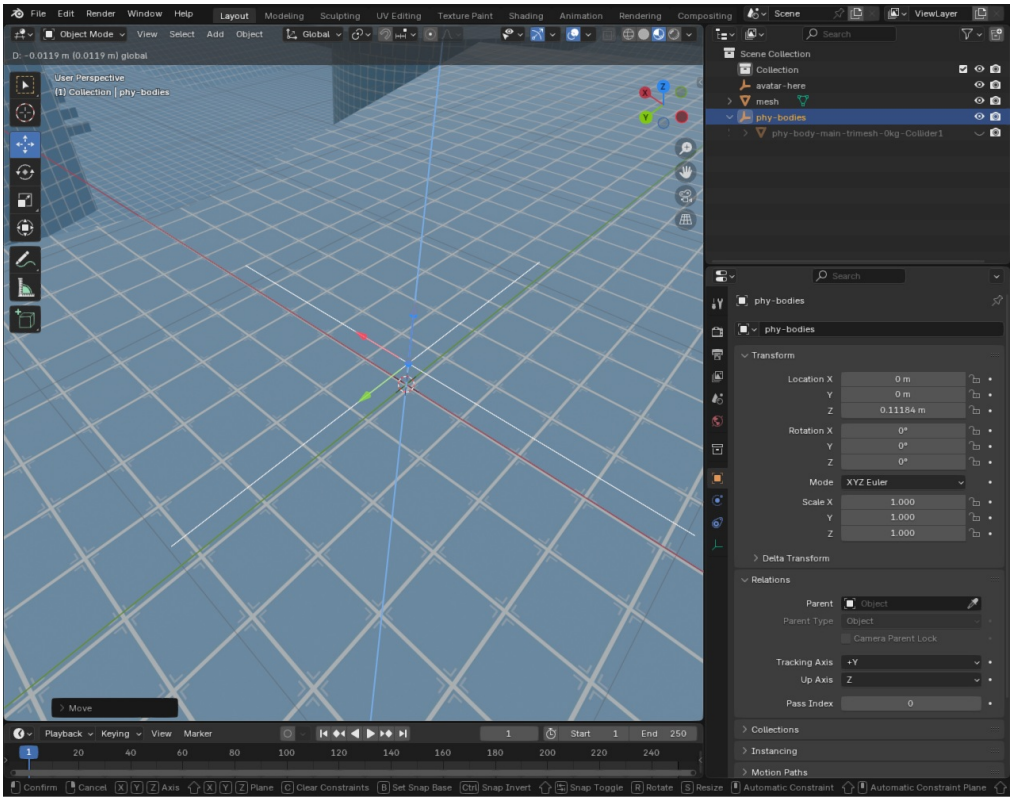


Figure 9.5 – Avatar position “**`avatar-here`**”

9.7 Drag and Drop Loading Workflow

The drag and drop process follows a predictable sequence:

1. The user drags a GLTF file into the browser
2. The engine reads the GLTF structure
3. Visible meshes are added to the render scene
4. Collider meshes are registered with the physics system
5. The avatar-here plane is located
6. The avatar is repositioned accordingly
7. The level becomes interactive

This workflow allows contributors to focus on level design and structure rather than engine configuration.

9.8 Purpose of the Prototype Approach

By working with a simple three part prototype level, contributors can clearly understand how a GLTF asset functions as a complete virtual environment.

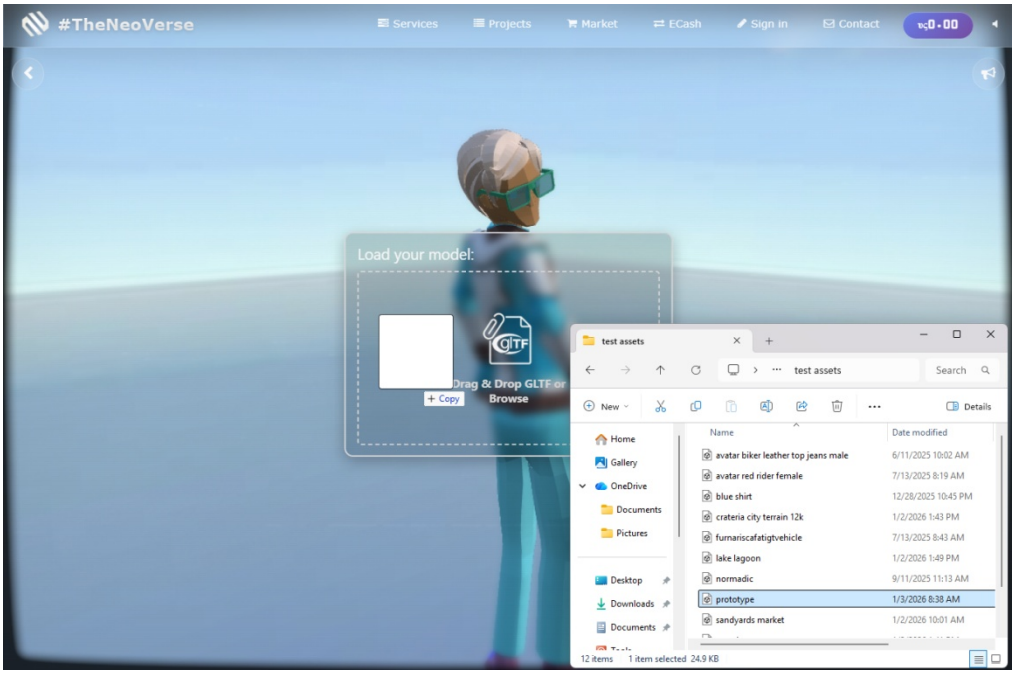


Figure 9.6 – "**`prototype.glb`**" is drag & dropped

This approach supports rapid experimentation, clear debugging, and a strong conceptual understanding of how visual meshes, physics colliders, and spatial anchors interact inside a browser based Virtual Experience Engine.

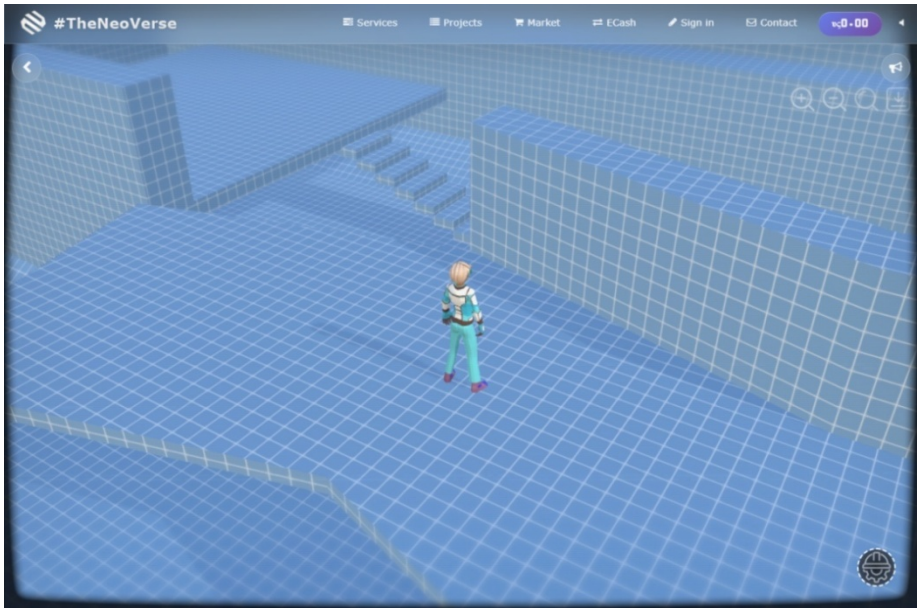


Figure 9.7 – "**`prototype.glb`**" is loaded

9.9 Summary

This chapter introduced the drag and drop workflow for loading levels into the Virtual Experience Engine. A simple prototype level was used to demonstrate the relationship between visible meshes, collider meshes, and the avatar-here plane.

By mastering this structure, contributors can quickly build, test, and iterate virtual worlds using GLTF assets without complex setup or tooling.

In the next chapter, interaction logic and object based behaviors will be introduced to further extend the capabilities of dynamically loaded levels.

10. Drag and Drop Avatar System
10.1 Introduction

In addition to loading environments through drag and drop, the Virtual Experience Engine also supports drag and drop avatars. An avatar represents the user inside the virtual world and defines how movement, camera behavior, and interaction are experienced.

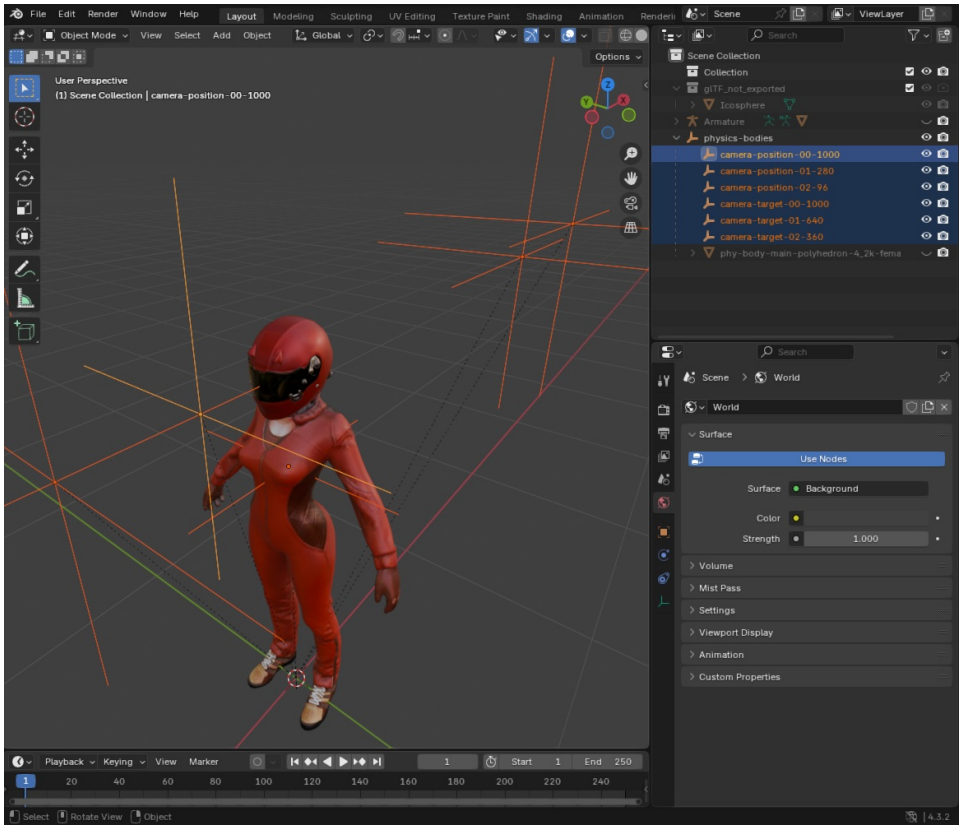


Figure 10.1 – “**`avatar red rider female.glb`**”

This chapter introduces a structured avatar prototype that can be loaded dynamically into the engine. The focus is on how an avatar GLTF asset defines camera behavior, view modes, and user interface attachment points using simple meshes and named plane axes.

10.2 Concept of a Drag and Drop Avatar

A drag and drop avatar is treated as a self contained GLTF asset. When the avatar is loaded into the environment, the engine binds camera systems, movement logic, and UI references directly to the avatar structure.

This approach allows different avatars to define their own camera behavior and interaction rules without modifying core engine logic. Designers can experiment with multiple avatar styles while maintaining consistent navigation and UI behavior.

10.3 Core Avatar Components

A basic avatar prototype is composed of three essential elements:

1. Camera plane axes that define camera positions and targets
2. A visible mesh that represents the avatar body
3. A UI reference plane axis that allows interface elements to follow the avatar

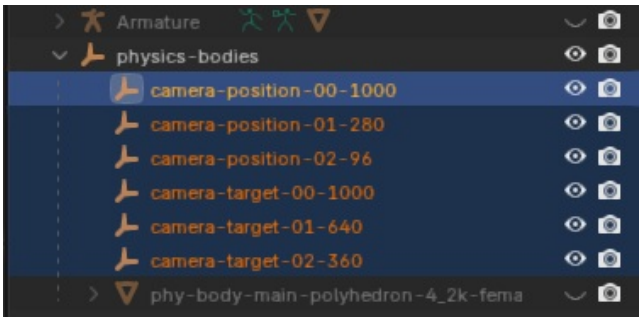


Figure 10.2 – Avatar components

Each element has a specific responsibility inside the engine.

10.4 Camera Plane Axes and View Modes

The avatar contains one or more named plane axes that define where the camera should move and what it should look at. These plane axes act as camera anchors rather than fixed camera positions.

When switching between camera modes, the camera position and its target smoothly interpolate toward the corresponding plane axes. This interpolation uses linear interpolation to avoid sudden jumps and preserve spatial continuity.

Common camera modes include:

First person view
The camera position interpolates to a plane axis located near the avatar head. The camera target interpolates forward to align with the avatar orientation.

Third person view
The camera position interpolates to a plane axis behind the avatar. The camera target interpolates toward the avatar body or chest area.

Far third person view
The camera position interpolates to a plane axis placed further behind or above the avatar. The camera target interpolates toward the full avatar body, allowing broader environmental awareness.

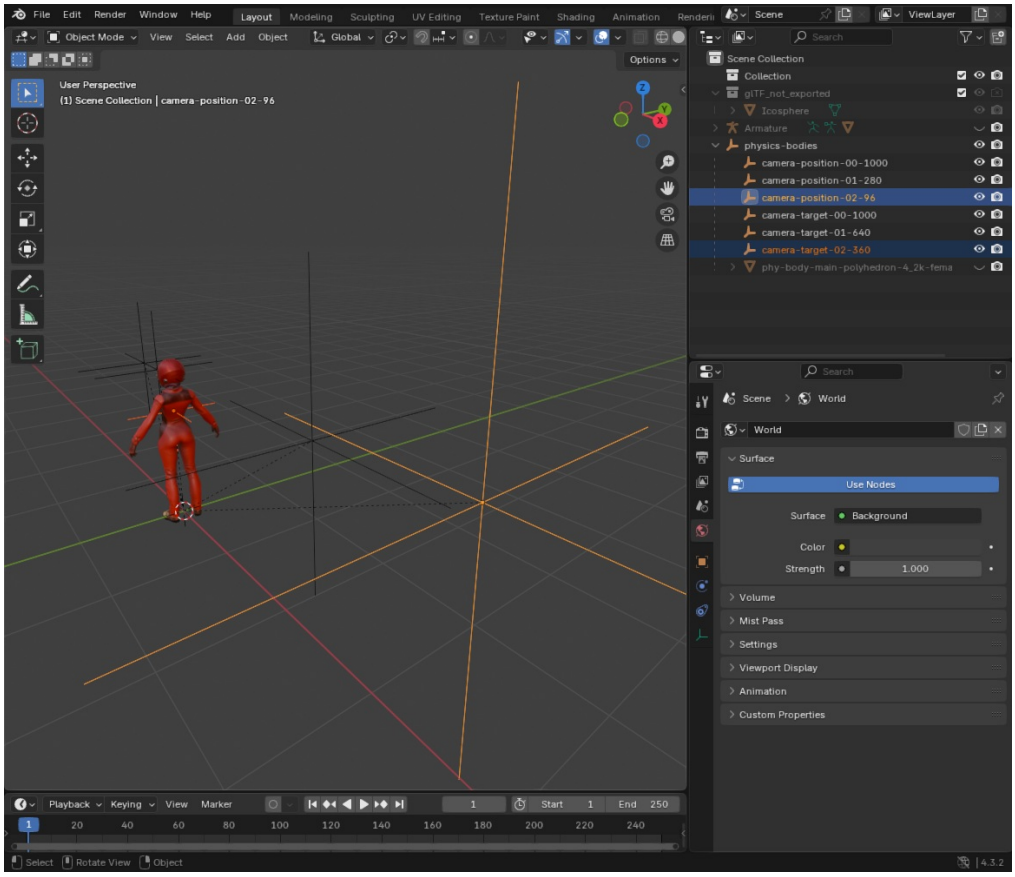


Figure 10.3 – Fat third person camera & target plane axises

Additional camera modes can be defined by adding more plane axes to the avatar GLTF. This allows flexible experimentation without changing engine code.

10.5 Camera Intersect Mesh

The avatar includes a simple mesh that represents the camera intersect region. This mesh is not intended for visual presentation. Its purpose is to assist with camera logic such as view obstruction, collision checks, or camera behavior tuning.

By using a dedicated intersect mesh, the engine can determine when the camera should adjust position to avoid clipping through geometry or entering restricted areas.

10.6 UI Information Plane Axis

Above the avatar, a plane axis is placed to serve as a reference point for user interface elements. This plane allows UI components to follow the avatar smoothly as it moves through the environment.

Typical uses include:

Name tags
Status indicators
Interaction icons
Information panels

This plane axis ensures that UI elements remain spatially anchored to the avatar rather than fixed to the screen.

10.7 Clickable Avatar UI and Modals

UI elements attached to the avatar information plane axis can be interactive. When clicked or tapped, these elements can trigger modal panels or overlays.

Examples of modal content include:

Avatar user information
Profile details
Interaction options
Contextual actions

This system allows avatars to function not only as navigational entities but also as interactive representations of users within the virtual environment.

10.8 Drag and Drop Avatar Loading Workflow

The avatar loading process follows a clear sequence:

1. The user drags an avatar GLTF file into the browser
2. The engine parses the avatar structure
3. Camera plane axes are detected and registered
4. The avatar mesh is added to the scene

5. The UI information plane axis is located
6. Camera and UI systems bind to the avatar
7. The avatar becomes active and controllable

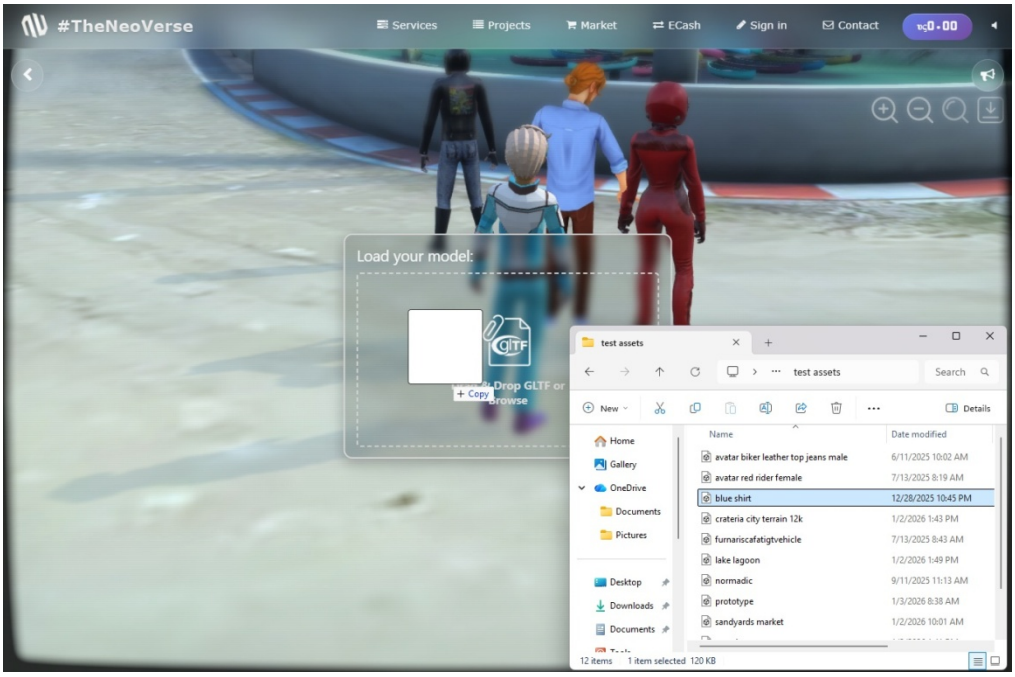


Figure 10.4 – Drag & dropping avatars

This workflow enables rapid testing of avatar designs and camera behaviors.

10.9 Purpose of the Avatar System

By defining camera behavior and UI attachment points directly inside the avatar GLTF, contributors gain full control over how users experience navigation and interaction.

This system supports experimentation with different camera styles, presentation modes, and social interaction features while keeping the engine modular and flexible.

10.10 Summary

This chapter introduced the drag and drop avatar system and its core components. Camera plane axes define multiple view modes through smooth interpolation. A camera intersect mesh supports stable camera behavior. A UI information plane axis allows interface elements to follow and interact with the avatar.

Together, these elements enable dynamic, customizable avatars that integrate seamlessly into browser based Virtual Experience Environments.

In the next chapter, we will explore object interaction, selection, and contextual behaviors driven by raycasting and input events.

11. Vehicle Drag and Drop into the Virtual Experience Engine
11.1 Introduction

Vehicles extend the capabilities of a Virtual Experience Engine beyond walking scale navigation. By supporting drag and drop vehicles, the engine allows contributors to introduce mobility systems such as scooters, cars, trucks, and specialized vehicles without modifying core engine logic.

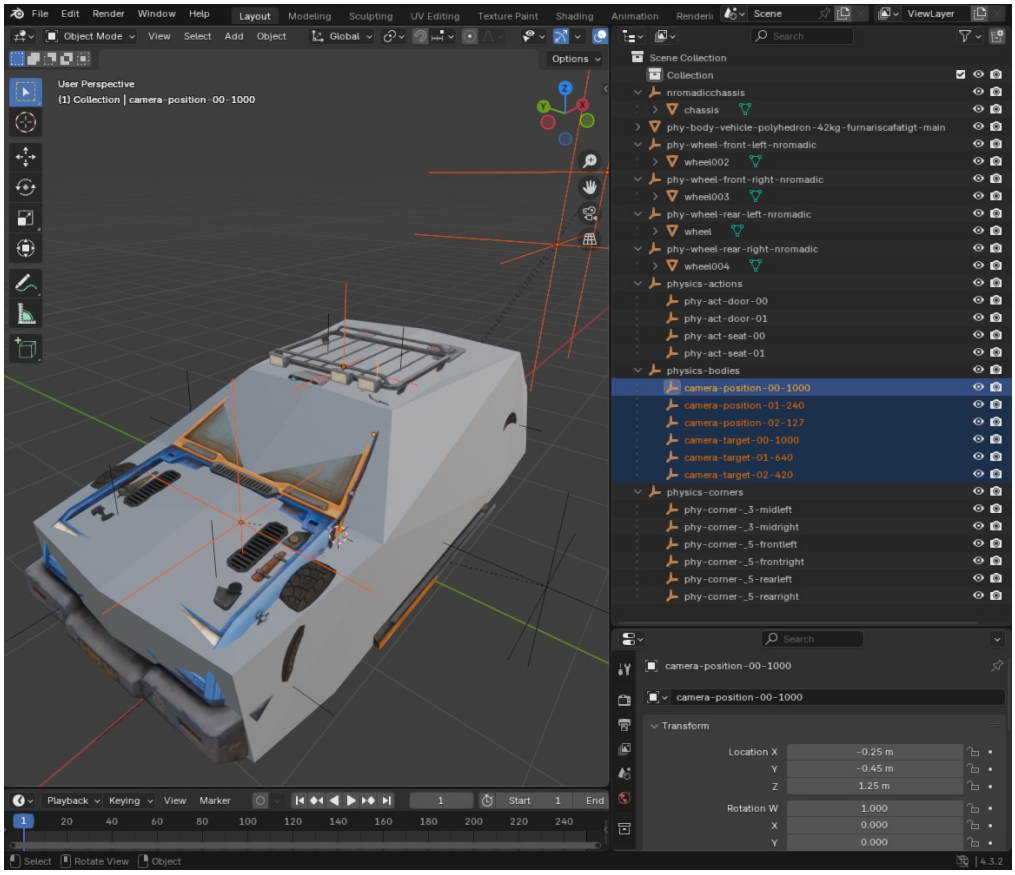


Figure 11.1 – “**`normadic.glb`**”

This chapter introduces the structure of a drag and drop vehicle and explains how vehicle behavior is defined using simple meshes and named plane axes inside a GLTF asset. The vehicle system follows the same design philosophy as the avatar system, prioritizing clarity, modularity, and real time experimentation.

11.2 Concept of a Drag and Drop Vehicle

A vehicle is treated as a self contained GLTF asset. When a vehicle is dragged and dropped into the browser based Virtual Experience Engine, the system detects its internal structure and binds movement, camera, and interaction logic automatically.

This approach allows designers and engineers to prototype different vehicle types using familiar modeling tools such as Blender, while maintaining consistent runtime behavior inside the engine.

11.3 Vehicle Components Overview

A basic vehicle prototype consists of several functional components that can be inspected and edited directly inside Blender or any compatible 3D modeling tool.

Core vehicle components include:

Camera plane axes
Seat plane axes
Door plane axes
Chassis collider
Wheel meshes and wheel axes

Each component defines a specific aspect of vehicle behavior and interaction.

11.4 Vehicle Camera Plane Axes

Vehicles define their camera behavior using plane axes similar to those used by avatars. These plane axes act as camera anchors rather than fixed camera positions.

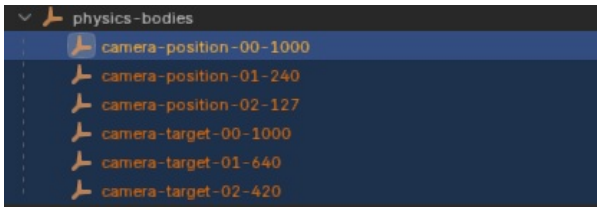


Figure 11.2 – Camera axises

Common camera plane axes include:

Driver view camera
Third person chase camera
Far third person camera

When the user enters a vehicle, the camera position and target smoothly interpolate toward the appropriate plane axes. This allows consistent camera transitions across different vehicle types while preserving flexibility in camera placement.

Additional camera modes can be added by defining more camera plane axes inside the vehicle GLTF.

11.5 Seat Plane Axes

Seat plane axes define where avatars are positioned when entering a vehicle. Each seat plane axis represents a valid seating location such as:

Driver seat
Passenger seat
Rear seat

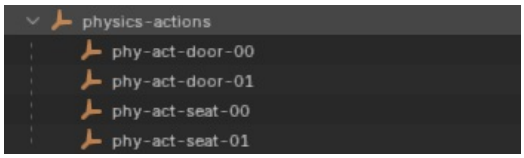


Figure 11.3 – Action axises

When an avatar enters a vehicle, the engine aligns the avatar to the corresponding seat plane axis. This ensures consistent positioning and orientation regardless of vehicle scale or layout.

Multiple seat plane axes allow vehicles to support multiple occupants.

11.6 Door Plane Axes

Door plane axes define interaction points for entering and exiting the vehicle. These plane axes are used to determine where an avatar should stand when interacting with a vehicle door.

Door plane axes support behaviors such as:

Entry and exit positioning
Door interaction triggers
Seat assignment logic

By defining door locations inside the vehicle GLTF, designers can customize interaction flow without modifying engine code.

11.7 Chassis Collider

The chassis collider defines the physical body of the vehicle. This collider is typically a simple box or basic convex shape that approximates the overall volume of the vehicle.

The chassis collider is used for:

Collision detection
Physics simulation
Vehicle stability

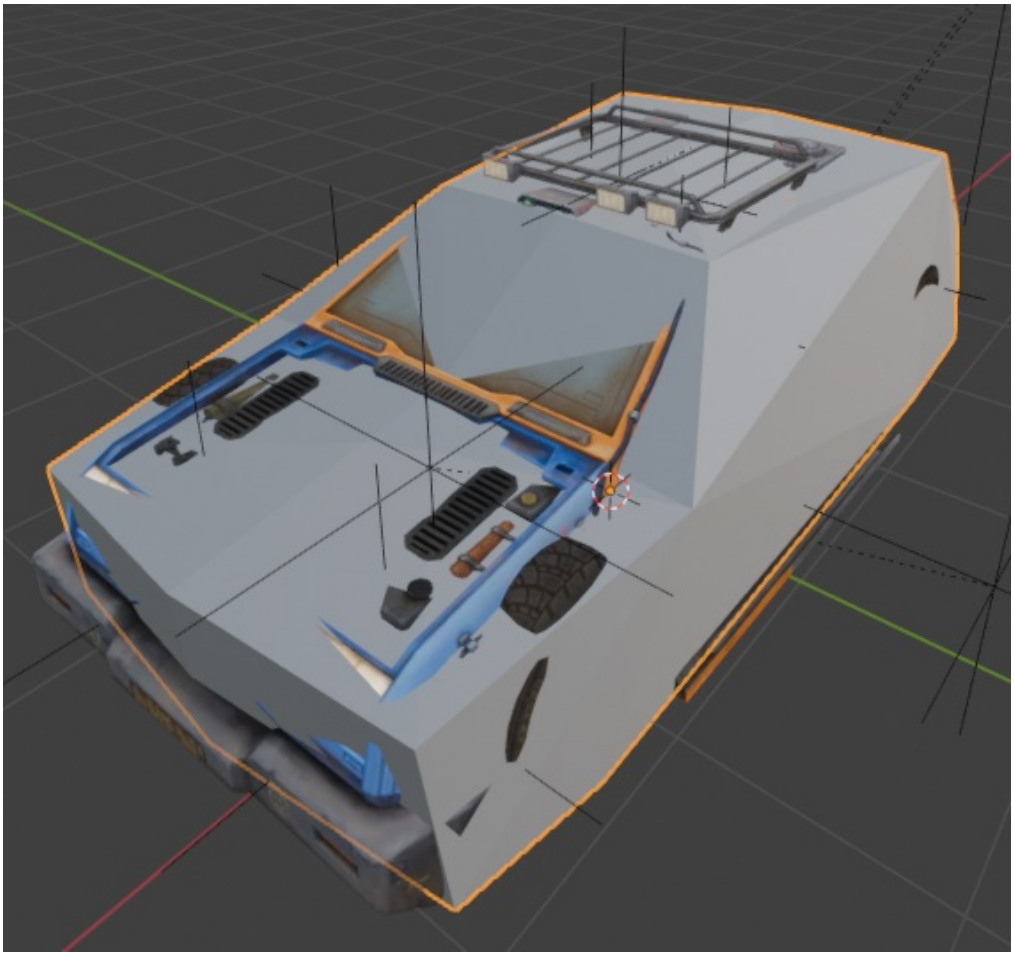


Figure 11.4 – Chassis collider “**`phy-body-vehicle-polyhedron-42kg-normadic`**”

Using a simplified collider improves performance and prevents unstable physics behavior caused by complex geometry. Visual detail remains separate from physical representation.

11.8 Wheels and Wheel Axes

Wheels are aligned using named wheel axes inside the vehicle GLTF. Each wheel mesh is positioned and oriented relative to its corresponding wheel axis.

Figure 11.5 – Wheel axises

Wheel configuration varies depending on vehicle type:

Two wheeled vehicles such as scooters or motorcycles use front and rear wheel axes Four wheeled vehicles such as cars use front left, front right, rear left, and rear right wheel axes. Six wheeled vehicles such as trucks may include additional rear wheel axes. Specialized vehicles such as space rovers may define unique wheel layouts and counts

By reading wheel axes dynamically, the engine supports a wide range of vehicle configurations without hardcoded assumptions.

11.9 Vehicle Loading Workflow

The drag and drop vehicle loading process follows a structured sequence:

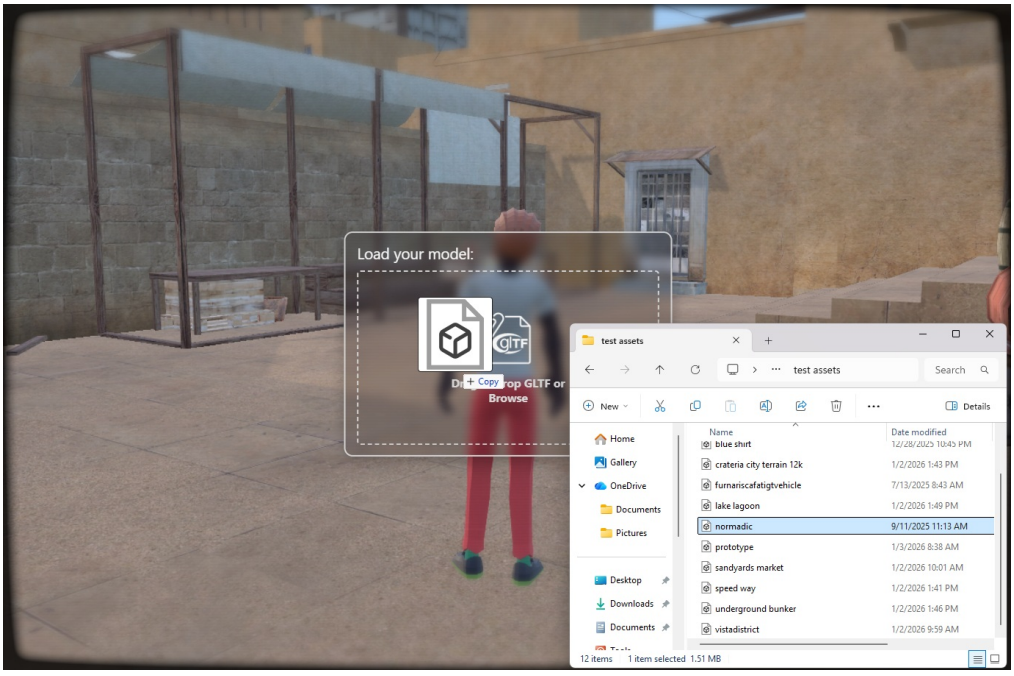


Figure 11.6 – Drag & dropping “**`normadic.glb`**”

1. The user drags a vehicle GLTF file into the browser
2. The engine parses the vehicle structure
3. Camera plane axes are detected and registered
4. Seat and door plane axes are identified
5. The chassis collider is created
6. Wheel axes and wheel meshes are aligned
7. The vehicle becomes interactive

Figure 11.7 – Vehicle is loaded into the scene

This workflow allows rapid testing of vehicle designs and configurations in real time.

11.10 Purpose of the Vehicle System

By defining vehicle behavior directly inside the GLTF asset, contributors gain full control over vehicle layout, interaction points, and camera behavior.

This system supports rapid prototyping of transportation systems, exploration vehicles, and interactive simulations while keeping the Virtual Experience Engine modular and scalable.

11.11 Summary

This chapter introduced drag and drop vehicles within the Virtual Experience Engine. Vehicles are defined using camera plane axes, seat and door plane axes, a simplified chassis collider, and flexible wheel axis configurations.

By following this structure, designers and engineers can create a wide variety of vehicle types, from simple scooters to complex multi wheel or off world vehicles, and test them instantly inside a browser based virtual environment.

In the next chapter, advanced interaction systems such as vehicle entry logic, camera transitions, and multi user synchronization can be explored.

12. Advantages of the Virtual Experience Engine

12.1 Modular Pipelines

The Virtual Experience Engine separates scene management, physics, and rendering into independent pipelines. This separation enables parallel execution and improves overall system stability.

Key advantages include:

Parallel processing for smoother performance, Simplified debugging and faster iteration due to decoupled systems, Scalable architecture where new features can be added without disrupting existing functionality

Example
Updating physics calculations for a crowd simulation does not interrupt rendering or camera movement.

12.2 Browser Based Accessibility

The Virtual Experience Engine runs entirely inside a standard web browser.

Benefits include:

No software installation required
Compatibility with laptops, tablets, and low end devices
Instant project sharing through simple browser links

Example
A design team can share a live BIM walkthrough with a client who can explore the space interactively without installing any software.

12.3 Asynchronous Physics with Web Workers

Physics calculations are executed independently from rendering using Web Workers.

This approach allows:

Stable frame rates during heavy simulations, Support for large numbers of physics bodies such as furniture, vehicles, and dynamic environments, Smooth user interaction even in complex scenes

Example
In an exhibition hall simulation, hundreds of interactive objects can coexist without degrading performance.

12.4 LOD and Metis Optimization

The engine uses dynamic mesh and texture optimization strategies.

These include:

Automatic level of detail reduction for distant objects
Mesh simplification through Metis optimization
Lower memory usage while preserving visual quality

Example
In large city environments, distant buildings are rendered with simplified geometry while nearby structures retain detail.

12.5 Rapid Prototyping

The Virtual Experience Engine enables fast design iteration directly in the browser.

Advantages include:

Immediate visual feedback during development
Reduced turnaround time between design and revision
Improved collaboration during live review sessions

Example
Interior designers can modify furniture layouts in real time during client presentations.

12.6 Limitations Compared to Traditional Engines

Despite its strengths, the Virtual Experience Engine has limitations when compared to desktop based engines.

Graphics fidelity
Browser based rendering may not match the highest quality visuals achievable with engines such as Unreal.

Complex simulations
Advanced artificial intelligence systems and deep physics simulations are better handled by traditional engines.

Platform specific VR and AR
Full scale VR and console deployments often require Unity or Unreal.

12.7 Comparative Use Case

Scenario
A city planner presents a proposed urban development to stakeholders.

Using the Virtual Experience Engine:

Interactive browser based city model
Real time pedestrian and traffic simulation
Immediate stakeholder feedback

Using traditional engines:

Higher visual fidelity
Platform specific builds and installations
Longer iteration cycles

Outcome
The Virtual Experience Engine prioritizes accessibility and collaboration, while traditional engines focus on high end visual output.

12.8 Summary

The Virtual Experience Engine offers a modular, browser based architecture optimized for accessibility, collaboration, and rapid iteration.

It is well suited for architecture, urban planning, interior design, events, and product visualization. Traditional engines remain better suited for high fidelity rendering and complex game development scenarios.

13. Modular Pipeline with Asynchronous Physics
13.1 Introduction

This chapter explores the internal architecture of the Virtual Experience Engine. It explains how modular pipelines and asynchronous physics systems work together to deliver high performance interactive experiences in the browser.

13.2 Pipeline Separation

The engine is divided into distinct pipelines:

Scene management
Rendering
Physics
Input handling

Each pipeline operates independently and communicates through well defined interfaces.

This separation prevents heavy computations in one system from blocking others.

13.3 Parallel Execution Model

Rendering focuses solely on visual updates and frame presentation.
Physics runs asynchronously using Web Workers.
Scene logic and input handling operate on the main thread with minimal blocking.

This parallel model ensures consistent frame rates even under complex simulation conditions.

13.4 Physics Decoupling Strategy

Physics calculations are time stepped independently from rendering.

Key benefits include:

Predictable simulation behavior
Stable movement and collision handling
Improved responsiveness during interaction

Physics results are synchronized back to the rendering pipeline without stalling visual updates.

13.5 Practical Impact on Virtual Experiences

This architecture enables:

Large scale environments
Multiple moving entities such as avatars and vehicles
Interactive elements without frame drops

Designers and developers can focus on content creation rather than performance constraints.

13.6 Summary

By separating pipelines and running physics asynchronously, the Virtual Experience Engine achieves smooth, scalable, and browser friendly performance.

This architecture forms the foundation for advanced systems such as drag and drop levels, avatars, vehicles, and multi user interaction.

14. Asynchronous Physics in Detail

Physics calculations can be CPU intensive, especially with multiple dynamic objects. The Virtual Experience Engine handles this with an asynchronous architecture:

- Physics computations are moved off the main thread using **Web Workers**
- VEE uses **Overdriven CannonJS**, an enhanced version of CannonJS optimized for large numbers of rigid bodies
- Physics updates occur independently of rendering, preventing frame drops

Example
A large exhibition space with 100 chairs and 20 interactive displays:

- The physics module calculates collisions and dynamics in the background
- The scene and rendering modules remain fully responsive, allowing smooth camera movement and user interactions

14.1 Advantages of This Pipeline

1. **High Performance**: Parallel processing prevents slowdowns during complex simulations
2. **Smooth Interactivity**: Browser-based interactions remain fluid even with many physics objects
3. **Dynamic Optimization**: LOD and Metis mesh crunching allow high detail models to run on mid-range hardware
4. **Scalability**: New modules, such as AI or networking, can be added without altering the core pipelines

15. Asset Optimization and Mesh Management

Efficient asset and mesh management is critical to maintain smooth performance in browser-based Virtual Experience Engines. Large-scale architectural models, urban environments, or high-poly product models can quickly overwhelm the GPU or CPU if not optimized.

This chapter explains how to prepare, optimize, and manage assets for high fidelity without compromising performance.

15.1 Mesh Optimization Techniques

1. Polygon Reduction

- Reduce polygons in complex models using **Metis mesh crunching**
- Maintain visual fidelity by selectively reducing detail on distant or static objects

Example: A highly detailed BIM staircase may be reduced from 50,000 to 10,000 polygons while preserving its shape for distant camera views

2. Level of Detail (LOD)

- Create multiple mesh versions at different resolutions
- The engine switches automatically based on camera distance

- Far away objects are rendered with fewer polygons to save processing power

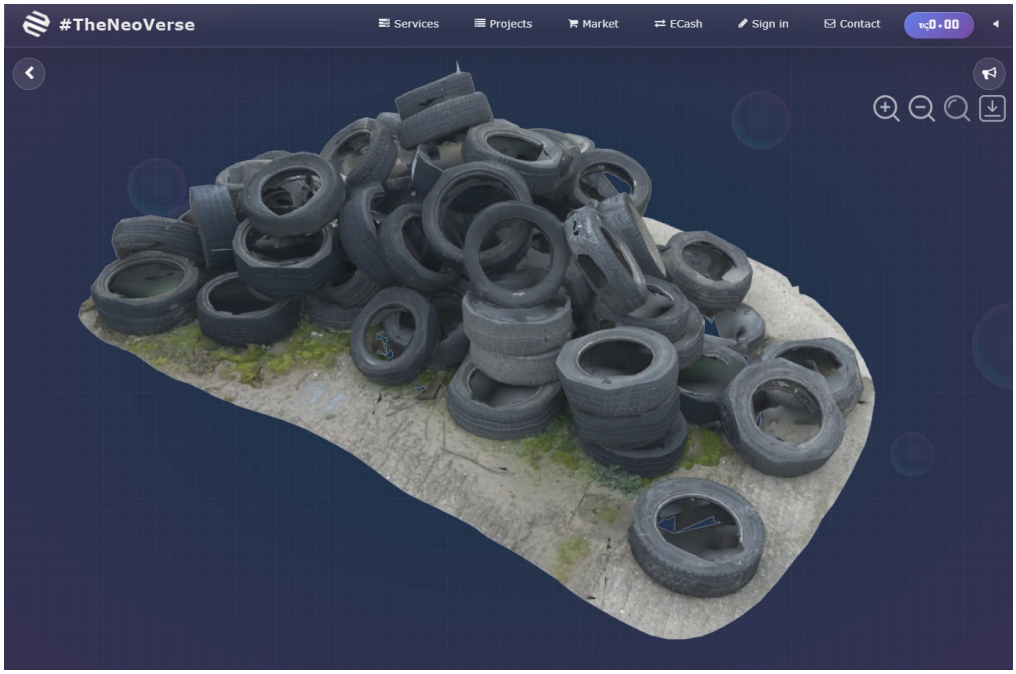


Figure 15.1 – LOD demo
"https://theneoverse.web.app/#threeviewer&&load=12"

Example: Trees, vehicles, or furniture appear simpler when viewed from a distance

3. Mesh Instancing

- Reuse a single mesh multiple times without duplicating geometry data
- Ideal for repeating objects like chairs, lamps, or cubicles
- Reduces memory consumption and improves frame rates

4. Animated Mesh Optimization

- Optimize skeletal or morph animations by reducing bone influences
- Pre-bake animations where possible to minimize runtime calculations

15.2 Texture and Material Management

1. Texture Atlases

- Combine multiple small textures into a single large texture to reduce draw calls and GPU overhead

2. Mipmapping

- Automatically generate smaller versions of textures for objects at a distance
- Reduces aliasing and improves rendering speed

3. Physically Based Rendering (PBR)

- Ensure materials respond realistically to light while balancing performance
- Simplified shaders can be used for distant objects to reduce calculations

4. Texture Compression

- Use compressed formats like JPEG, PNG, or KTX2 for WebGL/WebGPU compatibility
- Reduces memory usage and download times for large scenes

15.3 Scene Graph and Asset Management

1. Hierarchical Scene Graphs

- Organize objects in a tree structure for efficient updates
- Group related objects together (e.g., all furniture in a room)
- Facilitates selective updates and culling

2. Memory Management

- Dispose of unused geometries, textures, and materials to free GPU memory
- Track asset references to prevent memory leaks in long-running sessions

3. Dynamic Loading (Streaming Assets)

- Load assets on demand based on camera location or user interaction
- Essential for large urban or BIM models where loading everything at once is prohibitive

15.4 Example Workflow: Optimizing a Multi-Story Building

1. **Model Preparation**

- Import building from 3ds Max or Revit
- Reduce high-poly furniture meshes using Metis crunching

2. **Texture Setup**

- Combine interior textures into atlases
- Apply PBR materials for realistic lighting

3. **LOD Creation**

- Create three levels of detail for exterior and interior elements
- Far-away floors switch to low-poly meshes automatically

4. **Dynamic Loading**

- Load only floors within the camera's immediate vicinity
- Stream in additional floors as the user navigates

5. **Performance Testing**

- Monitor FPS and memory usage
- Adjust mesh simplification and texture resolution as needed

15.5 Key Takeaways

- Effective asset optimization is essential for smooth, real-time experiences
- Use LOD, mesh instancing, and Metis mesh crunching for performance without sacrificing visual quality
- Texture atlases, mipmapping, and PBR materials ensure realism and efficiency
- Dynamic loading and careful scene graph management allow massive BIM and urban models to run smoothly in browsers

16. Performance Optimization Techniques

High-fidelity interactive experiences in Virtual Experience Engines require careful performance management. Browser-based 3D engines like ThreeJS + CannonJS rely on the user's hardware, so optimization ensures smooth frame rates, responsive interactions, and real-time feedback even with complex scenes.

16.1 Why Performance Optimization Matters

1. **Smooth User Experience**

- Users expect 30–60 FPS in interactive applications
- Lag, stutter, or delayed interactions break immersion and usability

2. **Hardware Limitations**

- VEEs run in browsers across devices with varying GPU/CPU power
- Optimization enables accessibility without requiring high-end hardware

3. **Complexity of Real-Time Scenes**

- Large BIM models, dense urban environments, and high-poly objects increase computational load
- Physics, lighting, and dynamic interactions add further processing requirements

16.2 Asynchronous Processing

1. **Decoupling Physics from Rendering**

- Physics calculations run in parallel to rendering using Web Workers
- Heavy simulations such as rigid bodies and collisions do not block the main render loop

2. **Benefits**

- Reduces frame drops and improves responsiveness
- Allows complex scenes with many dynamic objects to remain interactive

16.3 Level of Detail (LOD) Management

1. **Concept**

- Objects have multiple mesh resolutions
- The engine dynamically swaps meshes based on camera distance

2. **Implementation**

- High-resolution mesh when the camera is close
- Medium-resolution mesh at mid-range
- Simplified mesh or impostor at far distance

3. **Impact**

- Reduces GPU workload
- Maintains visual fidelity where it matters most

16.4 Mesh Optimization Techniques

1. **Metis Mesh Crunching**

- Reduces polygon count without visibly affecting quality
- Processes static and dynamic objects to minimize memory usage

2. **Batching**

- Groups multiple static meshes into a single draw call

- Reduces overhead and GPU state changes

3. **Instancing**

- Reuses a single geometry across many objects with different positions or materials
- Ideal for trees, street lamps, chairs, or repeated architectural elements

16.5 Texture Optimization

1. **Texture Atlases**

- Combine multiple textures into one large image to reduce material swaps

2. **Mipmapping**

- Automatically adjusts texture resolution based on object distance
- Reduces aliasing and memory bandwidth usage

3. **Compression**

- Use compressed formats such as JPEG, PNG, WebP, or KTX2 to reduce memory footprint

16.6 Frustum Culling

1. **Concept**

- Only render objects visible in the camera's view frustum

2. **Benefits**

- Avoids rendering objects outside the user's view
- Reduces GPU load and increases frame rate

16.7 Occlusion Culling

1. **Concept**

- Avoid rendering objects hidden behind other geometry

2. **Implementation**

- Use bounding volumes or hierarchical z-buffer techniques
- Particularly useful in dense interiors, urban environments, or complex BIM models

16.8 Physics Optimization

1. **Selective Updates**

- Only update active physics bodies or objects in motion

2. **Simplified Collision Shapes**

- Use primitive shapes such as boxes, spheres, or capsules instead of high-poly meshes

3. **Sleeping Objects**

- Objects at rest are “put to sleep” and do not consume CPU until reactivated

16.9 Memory Management

1. **Asset Streaming**

- Load assets progressively based on proximity or user interaction

2. **Garbage Collection**

- Dispose of unused geometries, textures, and materials to free memory

3. **Resource Pooling**

- Reuse temporary objects to reduce allocation overhead

16.10 Performance Monitoring

1. **Metrics to Track**

- FPS (Frames per Second)
- Draw calls per frame
- Memory usage
- Physics simulation time

2. **Tools**

- Browser dev tools such as Chrome Performance tab
- Stats.js or similar real-time performance overlays

16.11 Best Practices

1. **Optimize Early**

- Implement LOD, batching, and culling during asset creation

2. **Keep Update Loops Lightweight**

- Avoid heavy computations in the main render loop

3. **Test on Target Devices**

- Ensure experience runs smoothly across desktops, laptops, and tablets

4. **Use Asynchronous Loading**

- Prevents blocking the main thread when loading large assets

16.12 Example Use Case: Optimizing a Multi-Story BIM Model

Scenario: A high-rise building with 50 floors, thousands of objects, and furniture

Steps:

1. Apply LOD for each furniture piece and architectural element
2. Use texture atlases for wall finishes, carpets, and flooring
3. Batch static elements per floor
4. Apply frustum culling and occlusion culling for non-visible areas
5. Run physics updates only on movable objects such as doors or elevators

Result: Smooth navigation at 60 FPS in a browser, even with high-fidelity details

16.13 Key Takeaways

- Browser-based VEEs require careful balancing of visual fidelity and performance
- Modular, asynchronous processing ensures smooth interactivity
- Mesh, texture, and physics optimizations allow complex scenes to run on standard hardware
- Monitoring and iterative refinement are crucial for scalable virtual environments

17. Deployment and Browser Compatibility

Deploying Virtual Experience Engines effectively requires understanding the constraints and capabilities of modern browsers, ensuring cross-platform accessibility, and implementing strategies for progressive loading and interactive performance. This chapter explores the methods, best practices, and tools necessary for successfully deploying VEEs built with ThreeJS + CannonJS.

17.1 Why Deployment Matters

1. **Cross-Platform Accessibility**

- VEEs are designed to run in browsers across desktop, laptop, tablet, and mobile devices
- Proper deployment ensures consistent performance and visuals without platform-specific installations

2. **Ease of Sharing and Collaboration**

- Browser-based delivery allows stakeholders to interact with 3D content immediately via a URL
- Eliminates software licenses, system requirements, or lengthy build processes

3. **Maintaining Performance Across Devices**

- Deployment strategies must consider varying GPU and CPU capabilities
- Techniques such as progressive loading, LOD, and texture optimization are critical

17.2 Cross-Browser Support

- **Supported Browsers:** Chrome, Firefox, Edge, Safari, Chromium-based browsers
- **WebGL vs WebGPU:**
    - WebGL: widely supported, stable
    - WebGPU: higher performance and advanced features but limited adoption
    - Use feature detection to dynamically select the best API
- **Polyfills and Fallbacks:** ensure basic functionality across all devices

17.3 Progressive Loading of Assets

- Load critical assets first, defer secondary ones
- Reduces initial load time while providing an interactive environment
- Techniques include lazy loading distant objects, streaming textures, and incremental asset bundles

17.4 Hosting and Delivery Options

- **Static Web Hosting:** GitHub Pages, Netlify, Vercel
- **Cloud Services:** AWS S3 + CloudFront, Google Cloud Storage, Azure Blob Storage
- **Enterprise Deployment:** Internal servers integrated with corporate intranets

17.5 Embedding VEEs into Websites

- **iFrame Integration:** embed scenes for client presentations
- **Direct API Integration:** expose the VEE as a module for dynamic control
- **Responsive Design:** adaptive camera and UI adjustments for different screen sizes

17.6 Performance Considerations for Deployment

- Asset compression, minification, caching, and lazy initialization prevent blocking the main thread

17.7 Testing Across Devices

- Desktop: ensure accessibility for mid-range machines
- Tablets and Laptops: manage LOD and simplify materials for consistent frame rates
- Mobile: implement touch navigation and optimize for limited GPU/CPU

17.8 Security and Permissions

- Use HTTPS and configure CORS for remote assets
- Request user consent for VR/AR hardware, camera, and motion sensors

17.9 Deployment Checklist

- Confirm cross-browser compatibility
- Optimize assets for progressive loading
- Implement caching and compression
- Test interactive features across devices
- Verify responsive UI
- Ensure security protocols are in place
- Validate physics and performance metrics

17.10 Example Use Case: Deploying a BIM Walkthrough

- Scenario: Multi-story office building with furniture, lighting, HVAC simulations
- Steps: compress textures, split floors into LOD meshes, defer secondary assets, test across browsers, embed in website, monitor performance
- Result: Interactive BIM walkthrough accessible on desktop and tablet without installations

17.11 Key Takeaways

- Deployment requires optimization, testing, and accessibility planning
- Progressive loading and modular asset management reduce load times
- Browser compatibility ensures VEEs are widely accessible

- Proper deployment maximizes VEE benefits for real-world projects

18. Case Studies and Example Projects

This chapter provides real-world examples of Virtual Experience Engines applied to architectural, urban, interior, event, and product design. Each case study highlights techniques, best practices, and lessons learned for interactive 3D experiences.

18.1 BIM Walkthrough: Multi-Story Office Building

- Fully explorable BIM model with structural, mechanical, and interior elements
- Interactive material switching and real-time lighting adjustments
- Physics-based interactions for movable furniture and doors
- Clients explore the building in a browser; designers receive instant feedback

18.2 Interactive Urban Plan: City Development

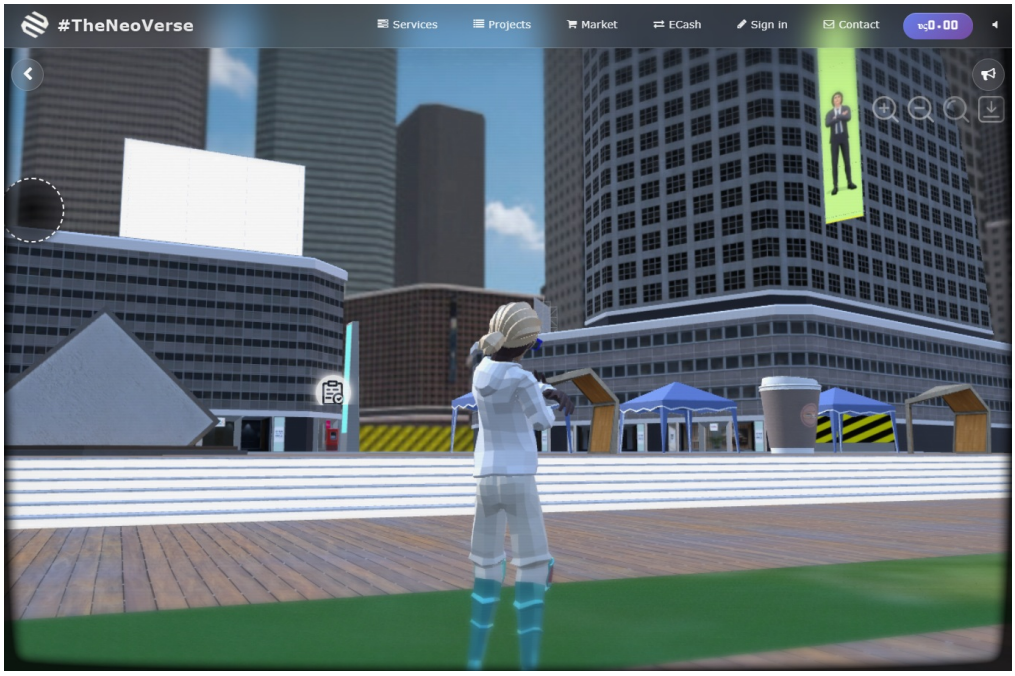


Figure 18.1 – Urban Design "https://theneoverse.web.app/#serini&&serini"

- Real-time navigation of streets, parks, and buildings
- Pedestrian and vehicle flow simulations
- Zoning, sunlight analysis, and dynamic environmental conditions
- Stakeholders can make informed urban design decisions in real-time

18.3 Interior Design: Residential Apartment

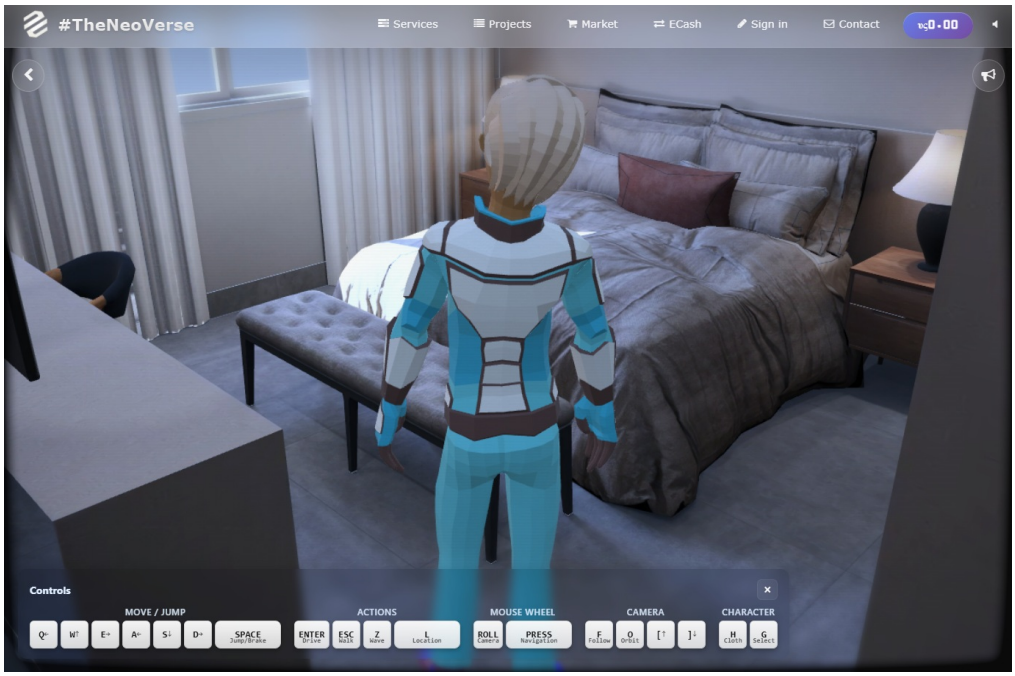


Figure 18.2 – Apartment Interior
"https://theneoverse.web.app/#threeviewer&&suite"

- Free-fly and orbit camera modes
- Real-time lighting adjustments and material swapping
- Physics-enabled objects for realistic interaction
- Clients explore design options and provide feedback, reducing rework

18.4 Event Space Layout: Conference Hall

- Real-time walkthrough with crowd simulation
- Interactive object placement for booths, stages, or furniture
- Trigger zones for audio, lighting, and animated effects
- Organizers optimize layouts and test emergency evacuation routes virtually

18.5 Product Visualization: Furniture Configurator

- Real-time material and color switching
- Interactive scaling, rotation, and physics-based interactions
- Mobile-friendly navigation for remote clients
- Clients make informed design decisions before production

18.6 Lessons Learned from Case Studies

- Modularity is key: separate scene, physics, and rendering pipelines
- LOD and optimization balance high-detail assets with browser performance
- Interactivity enhances user understanding
- Cross-platform testing ensures accessibility
- Browser-based platforms enable rapid iteration

18.7 Example Use Case Integration

- Combine urban planning and BIM walkthroughs for city-scale development
- Integrate interior visualization with event space planning
- Use product configurators for marketing or virtual showrooms

19. UI/UX for Interactive Virtual Environments

Creating compelling Virtual Experience Engines (VEEs) requires not only technical skill but also careful attention to user interface (UI) and user experience (UX). This chapter covers designing intuitive, immersive, and accessible interfaces that enhance user interaction in 3D environments.

19.1 Principles of UI/UX in VEEs

Key design principles:

- **Clarity:** Users should immediately understand interactive elements, controls, and feedback
- **Consistency:** Keep interaction patterns uniform to reduce learning curves
- **Affordance:** Interactive objects should indicate how they can be used (buttons, levers, draggable objects)
- **Responsiveness:** Provide immediate visual or auditory feedback to actions
- **Immersion:** UI should complement the 3D environment without obstructing presence

19.2 Types of User Interfaces

1. **2D Overlays (HUDs):** Menus, toolbars, data readouts; semi-transparent or context-aware
2. **3D In-World Interfaces:** Panels, buttons, or interactive objects within the 3D scene
3. **Contextual Menus:** Appear near objects when needed to reduce clutter
4. **Voice and Gesture Controls:** Natural interactions in VR/AR; voice commands for navigation or scene adjustments

19.3 Designing for Navigation and Interaction

- Guided navigation with visual cues and waypoints
- Camera control feedback: orbit, free-fly, first-person with smooth transitions

- Interactive highlights: hover effects, outlines, or glows
- Tooltips and hints for contextual help

19.4 Accessibility Considerations

- Keyboard and gamepad support
- Color blindness and contrast-friendly palettes
- Scalable UI elements (font size, HUD scale)
- Audio feedback and subtitles for inclusivity

19.5 Prototyping and Testing UX

- Wireframes and mockups before implementation
- Iterative testing with real users
- Analytics tracking for user interaction patterns
- A/B testing to determine effective interfaces

19.6 UI/UX Tools for VEEs

- **3D UI Libraries:** ThreeJS UI extensions, BabylonJS GUI
- **Prototyping Tools:** Figma, Adobe XD, Sketch
- **Analytics Tools:** Google Analytics, Mixpanel, WebSocket logs

19.7 Example Projects

1. Interactive museum tours with 3D info panels and audio guides
2. Collaborative design review with interactive HUDs for annotations
3. Product configurator with real-time 3D feedback

Outcomes: Participants can design intuitive, immersive, and accessible interfaces, implement 2D and 3D UI elements, and prototype and refine UX effectively.

20. Integrating External Data (BIM, GIS, IoT)

Modern VEEs can connect and interact with real-world data. This chapter covers importing, managing, and utilizing BIM, GIS, and IoT data within VEEs.

20.1 Building Information Modeling (BIM) Integration

- Incorporate detailed building models for interactive exploration
- Data types: structural elements, HVAC, MEP, materials, annotations
- File formats: IFC, RVT, DWG, FBX
- Workflow: export, optimize, import, maintain metadata
- Applications: walkthroughs, clash detection, facility management

20.2 Geographic Information Systems (GIS) Integration

- Accurately represent urban/regional environments
- Data types: terrain, elevation, satellite imagery, zoning, demographics
- Workflow: acquire GeoJSON, Shapefiles, KML; transform coordinates; overlay on 3D terrains
- Applications: urban planning, environmental impact visualization, public engagement

20.3 Internet of Things (IoT) Integration

- Visualize live sensor data in real-time
- Data sources: temperature, occupancy, energy, traffic, industrial sensors
- Workflow: connect via API, MQTT, or WebSockets; map data to scene objects; visualize metrics dynamically
- Applications: smart building monitoring, factory floor visualization, real-time event spaces

20.4 Challenges and Best Practices

- Optimize performance using LOD and culling
- Ensure data accuracy and synchronization
- Secure live IoT data communication

20.5 Example Project: Smart Campus Simulation

- BIM: detailed building interiors
- GIS: campus terrain, roads, landscape
- IoT: live occupancy and energy usage
- Users navigate in real-time, observing interactive data

Outcomes: Participants can integrate BIM, GIS, and IoT, visualize complex data, and create real-time simulations.

21. Future Trends in Virtual Experience Engines

VEEs are evolving with hardware, software, and interactive technology advancements. This chapter explores emerging trends and methodologies.

21.1 WebXR and Immersive Experiences

- VR/AR experiences directly in browsers
- Applications: immersive architectural walkthroughs, AR overlays, hybrid simulations
- Enables fully immersive, device-agnostic experiences

21.2 Cloud-Based Rendering and Streaming

- Offload computation to cloud servers
- Stream compressed frames to clients
- Benefits: high-fidelity visuals on low-spec devices, multi-user collaboration

21.3 AI-Assisted Content Generation

- Automates mesh simplification, texture generation, scene optimization
- Applications: texture creation, automatic LOD, intelligent NPCs
- Designers focus on creativity while AI handles technical tasks

21.4 Real-Time Physics and Soft-Body Simulation

- Advanced physics like soft-body dynamics and fluids
- Applications: realistic curtains, vegetation, water, mechanical prototypes
- Technologies: WebAssembly, GPU acceleration, Web Workers

21.5 Multiuser and Collaborative VEEs

- Collaborative environments for multiple users in real-time
- Applications: global design reviews, urban planning workshops, training simulations
- Technologies: WebRTC, cloud networking, real-time object synchronization

21.6 Integration with IoT and Real-World Data

- VEEs become digital twins of real environments
- Applications: smart buildings, city simulations, interactive product experiences

21.7 Sustainability and Green Design

- Real-time environmental impact visualization and energy simulation
- Lifecycle assessment within interactive virtual environments

21.8 Cross-Platform and Device-Agnostic Experiences

- Seamless functionality across browsers, desktops, tablets, mobile, VR
- Technologies: PWAs, WebGPU/WebGL2, responsive 3D design

21.9 Example Scenario: Future Smart City Simulation

- BIM, GIS, IoT integration
- AI-generated crowd simulations
- Cloud rendering for high-fidelity visualization
- Multi-device collaborative access

21.10 Key Takeaways

Participants will understand:

- WebXR, cloud rendering, and AI trends shaping VEEs
- Real-time, multiuser collaborative environments
- VEEs as digital twins of real-world spaces
- Emerging technologies for immersive, interactive, data-driven visualization