Text Script for “Virtual Experience Engine”

Table of Contents

1: Title

ThreeJS + CannonJS
Virtual Experience Engine Script

This class is intended for architectural designers, developers, and engineers who are new to Virtual Experience Engines or who require a refresher on virtual experience architecture and workflows. The primary focus is on browser-based showcasing of architectural and design projects, including BIM models, urban designs, interiors, exteriors, event spaces, and product presentations.

The course uses a web-based engine built with ThreeJS and CannonJS as a practical reference implementation, but the concepts taught are applicable to other virtual experience engines such as Unreal, Unity, Godot, and similar platforms.

It is recommended that participants have general computer skills and basic knowledge of design modeling, texture baking, and various design tools such as 3ds Max, SketchUp, Blender, or similar applications. Basic coding experience is helpful but not required. Knowledge of architecture, spatial design, or 3D visualization is beneficial.

2: ThreeJS + CannonJS Overview
Virtual Experience Engine
Architecture Overview

3D Browser Game Engine Features
High performance attributed to:

- **Three.js WebGL WebGPU rendering library**
- **Cannon.js physics simulation library**
- **Assets mesh simplifier**
- **Animated texture management**
- **3d assets manangement**
- **Virtual camera controls**
- **Character and vehicle controller**
- **Character or model costomization**

The **Virtual Experience Engine** concept was inspired by early web-based 3D experiments such as **GitHub Sketchbook** and **Trigger Rally**, both ThreeJS projects demonstrating interactive 3D environments in the browser. These projects illustrated how designers and developers could quickly prototype and visualize interactive experiences using web technologies.

The architecture of the engine draws on lessons from these projects, particularly the way ThreeJS handles **scene graphs, meshes, cameras, and basic physics**, enabling browser-based experiences that are both performant and accessible. To

further optimize rendering, the engine incorporates **Metis mesh crunching**, a technique that reduces geometry complexity and memory footprint, much like **Unreal Engine's Nanite**, allowing high-detail models to be rendered efficiently in real time.

Additionally, the engine implements **Level of Detail (LOD) management** and **mipmapping-style virtualization**, dynamically adjusting mesh and texture resolution based on camera distance and screen size. This ensures smooth performance and consistent visual fidelity, even in large or complex scenes, by prioritizing resources for objects closest to the viewer while simplifying distant objects.

Early FPS-style examples highlighted the possibilities of real-time navigation, spatial interactivity, and physics-enabled objects, forming the conceptual foundation for the engine.

The Virtual Experience Engine emphasizes **modularity and flexibility**, allowing architectural designers, developers, and engineers to create rich, interactive visualizations. Core features such as scene management, asset handling, physics simulation, mesh optimization, and LOD control are designed to be intuitive, so participants can focus on **spatial storytelling and project presentation** rather than low-level coding.

By combining these ideas, the engine supports **rapid prototyping, browser-based showcasing of projects**, and cross-disciplinary collaboration between designers, developers, and engineers.

3: Comparison between Virtual Experience Engine and Other Engines
ThreeJS + CannonJS Program Architecture
Other Engine Program Comparison and Advatange

Let's compare the **Virtual Experience Engine** to other common game and visualization engines such as **Unreal, Unity, and Godot**.

The Virtual Experience Engine is designed with a **browser-based, modular architecture**, optimized for interactive architectural visualization, BIM, urban design, interiors, exteriors, event spaces, and product presentation. It separates **scene management, mesh handling, physics simulation, and rendering pipelines**, which allows multiple processes to run concurrently, increasing throughput, responsiveness, and scalability.

Other Engines

Traditional engines like Unreal or Unity use a more **monolithic architecture**, where rendering, physics, and scene management are tightly integrated. While this supports high-performance real-time graphics, it also requires **heavier hardware, longer build processes, and specialized programming expertise**. Optimizations such as LOD, mesh streaming, and texture management are implemented at a low level, often requiring significant developer intervention.

Monolithic pipeline integrating scene, rendering, and physics

- High computational requirements
- Sequential processing can limit flexibility
- Less accessible for browser-based deployment

a.) An asset or object is loaded into the engine memory.
b.) Rendering, physics, and scripts are processed sequentially in a single pipeline.
c.) Frame results are computed and sent to the GPU or display buffer.
d.) Next assets or frames are processed after previous processing completes.

Separate modular systems for scene, mesh, physics, and rendering

- Supports **concurrent processing** across modules
- Optimized for **browser deployment** and accessibility
- Includes **Metis mesh crunching**, **LOD management**, and **mipmapping-style virtualization** to maintain high performance with complex models

a.) An asset or mesh is loaded into the **scene management module**.
b.) While the asset is being rendered, the **physics module** updates object interactions and collisions simultaneously.
c.) **LOD management** adjusts mesh complexity and texture resolution dynamically based on camera distance.
d.) **Metis mesh crunching** reduces polygon count and memory usage without noticeable visual impact.
e.) The rendering module composes the frame in real-time, ready for display in the browser.
f.) The next frame begins processing immediately, enabling smooth, interactive experiences.

Additional Advantages of the Virtual Experience Engine:

- **Rapid Prototyping:** Designers and developers can see changes instantly in a browser without lengthy build processes.
- **Cross-Disciplinary Collaboration:** Architecture, design, and engineering teams can work together on the same scenes without deep coding expertise.
- **Scalability:** Supports both small-scale product presentations and large-scale urban or BIM projects.
- **Lightweight Deployment:** Runs on standard browsers, removing the need for high-end graphics cards or platform-specific builds.

Comparison Summary Table

Feature	Virtual Experience Engine	Traditional Engines
Target Platform	Browser (cross-platform)	Desktop, Console, Mobile
Rendering Optimization	Metis mesh crunching, LOD management, mipmapping-style virtualization	GPU pipelines, Nanite-like mesh streaming, LOD
Physics	CannonJS or lightweight physics	Built-in or third-party engines

Feature	Virtual Experience Engine	Traditional Engines
	engines	(PhysX, Havok)
Concurrency	Scene, physics, and rendering modules process simultaneously	Sequential monolithic pipeline
Ease of Use	Accessible to designers, developers, engineers; minimal coding	Requires programming expertise and engine-specific knowledge
Deployment	Instant browser preview and sharing	Build per platform; higher resource requirement
Use Case Focus	Architectural visualization, BIM, interiors, exteriors, event spaces, product presentations	Games, simulations, high-fidelity interactive experiences

This modular approach allows the Virtual Experience Engine to **deliver high-fidelity, interactive visualization in the browser**, while maintaining accessibility and fast iteration, in contrast to traditional engines that are optimized for performance but require specialized skills and resources.

4: Knowledge Check 1
Question: What makes the Virtual Experience Engine unique

1. Can run interactive 3D scenes directly in a browser?
2. Separates scene, physics, and rendering modules for **parallel processing**?
3. Uses **Web Workers** to handle heavy calculations concurrently?
4. Updates **physics bodies asynchronously**, detached from the ThreeJS render cycle?
5. Optimizes large models with **LOD management and Metis mesh crunching**?
6. Enables **rapid prototyping** for designers, architects, and engineers?

5: Modular Pipeline with Asynchronous Physics
Virtual Experience Engine Architecture
Parallel Processing, Overdriven CannonJS, and Optimization

- Traditional engines often process physics, scene updates, and rendering sequentially in a monolithic pipeline. This can create bottlenecks, especially in browser-based or lower-spec environments.
- The Virtual Experience Engine **modularizes** these systems into separate pipelines: **Scene Module, Physics Module, Rendering Module, and LOD/Optimization Module**.
- **Physics calculations run asynchronously** on **Web Workers**, completely **detached from the ThreeJS render loop**, so heavy physics workloads do not block frame rendering.
- **Overdriven CannonJS** is used to handle complex rigid body simulations, collisions, and forces, allowing large numbers of physics bodies to update simultaneously.

- **LOD management and Metis mesh crunching** dynamically adjust polygon counts and texture resolutions based on camera distance, improving performance without compromising visual fidelity.

Example of Asynchronous Modular Pipeline

Frame	Scene Module	Physics Module (Worker + Overdriven CannonJS)	Rendering Module	LOD/Optimization Module
F0	Load Asset A			
F1	Update Scene	Begin asynchronous physics update for Asset A		Preprocess LOD & mesh optimization
F2	Load Asset B	Continue physics updates independently	Render Frame 1	Apply LOD & Metis crunching
F3	Update Scene	Physics updates all active bodies asynchronously	Render Frame 2	Adjust textures & LOD dynamically
F4	Load Asset C	Physics calculations continue on worker thread	Render Frame 3	Mesh optimization & cleanup

Stepwise Explanation:

1. **Scene Module:**
    - Updates positions of objects, camera movement, user interactions, and environmental changes on the main thread.
2. **Physics Module:**
    - Runs asynchronously on **Web Workers**, fully detached from rendering.
    - Overdriven CannonJS calculates collisions, rigid body dynamics, and forces in parallel.
    - Can handle large numbers of physics bodies without slowing down the frame rate.
3. **Rendering Module:**
    - Composes and displays frames in the browser canvas.
    - Works continuously, independent of the physics computation.
4. **LOD & Optimization Module:**
    - Dynamically adjusts mesh complexity, polygon counts, and texture resolution based on camera distance.
    - Uses **Metis mesh crunching** to reduce memory usage while maintaining visual fidelity.

Advantages of the Virtual Experience Engine Compared to Other Engines:

- **Concurrent, Modular Pipeline:** Scene, physics, rendering, and optimization work simultaneously, maximizing throughput.
- **Asynchronous Physics:** Physics updates happen **detached from the render cycle**, eliminating bottlenecks.
- **High-Performance Physics:** Overdriven CannonJS allows more rigid bodies and collision calculations than standard CannonJS.

- **Browser-Friendly:** No need for installation, platform-specific builds, or high-end hardware.
- **Rapid Iteration:** Designers, architects, and engineers can preview and interact with changes instantly.
- **Dynamic Optimization:** LOD and Metis mesh crunching allow complex architectural or urban models to run smoothly in real time.

Summary:

The Virtual Experience Engine is a **browser-optimized, modular, and highly parallel system** for interactive architectural and design visualization. By separating scene, physics, and rendering pipelines and using **Web Workers**, **overdriven physics**, and **dynamic optimization techniques**, it delivers **high-fidelity, real-time experiences** that are accessible to both designers and developers, setting it apart from traditional monolithic engines like Unity or Unreal in the context of **web-based visualization and rapid prototyping**.