Immersive Systems I
Hardware & Software

Developing Immersive Applications

Created by: Chek Tien TAN

Learning Objectives:

- describe common software and hardware components across XR devices
- explain general system architecture for immersive applications
- describe the software component types in WebXR applications - differentiate device capabilities and form factors

Common Software Components

At its core, an immersive application is a real-time interactive simulation.

1. Rendering System
2. Physics System
3. Input Handler
4. Audio Processor
5. AI System
6. etc.

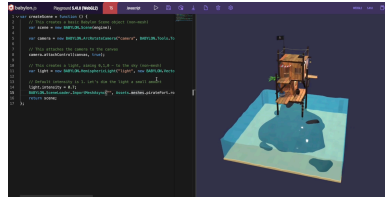
1. Rendering System

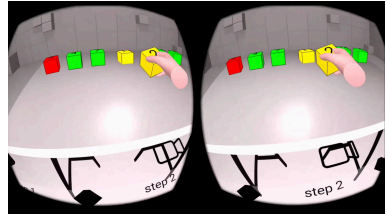
Responsibilities:

- Drawing graphics to the display
- Managing 3D models, textures, materials
- Configuring lighting and cameras
- Post-processing effects (bloom, DOF, etc.)
- **Stereo rendering** for VR

Under the Hood: Scene graph traversal, culling, clipping, shaders, rasterization

In WebXR: Babylon.js handles stereo rendering automatically.



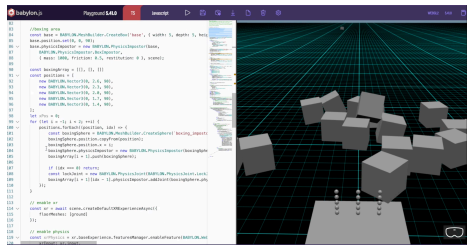


Video: https://youtu.be/OKD4jrnn4WE?t=1339

2. Physics System

Responsibilities:

- Collision detection and response
- Rigid body dynamics
- Gravity and forces
- Constraints and joints
- Raycasting for interaction



In WebXR: Babylon.js 6.0+ integrates Havok (recommended) via WebAssembly. Also supports Cannon.js and Ammo.js.

Video: https://youtu.be/OKD4jrnn4WE?t=1339

3. Input Handler

Responsibilities:

- Reading controller button presses
- Tracking hand/controller positions
- Processing gestures and interactions
- Managing input events
- Supporting multiple input devices

In WebXR: WebXR Device API provides standardized access to VR/AR controllers and hand tracking.

WebXR Input: XRSession & XRInputSource

Accessing XR input sources:

```
// Get XR session from Babylon.js
const xr = await scene.createDefaultXRExperienceAsync();
const session = xr.baseExperience.sessionManager.session; // XRSession

// Listen for input source changes (XRInputSource events)
session.addEventListener('inputsourceschange', (event) => {
for (const inputSource of event.added) { // inputSource is XRInputSource
  console.log('Input added:', inputSource.handedness); // 'left'/'right'
  console.log('Target ray mode:', inputSource.targetRayMode);
  }
});
```

Key WebXR input interfaces:

- `XRSession` — manages the XR experience lifecycle
- `XRInputSource` — represents controllers, hands, gaze
- `XRInputSourceArray` — collection of active inputs

https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API

4. Audio Processor

Responsibilities:

- Spatial audio, e.g., Head-Related Transfer Function (HRTF) models how sound waves interact with the head and ears to create realistic directional audio cues)
- Sound effects playback
- Music and ambient audio
- Audio attenuation with distance

In WebXR: Web Audio API provides spatial audio capabilities for immersive sound experiences.

https://doc.babylonjs.com/features/featuresDeepDive/audio/playingSoundsMusic

5. AI System

Responsibilities:

- NPC behavior and decision-making
- Pathfinding (A* algorithm, navigation meshes)
- Behavior trees for complex AI
- Procedural content generation
- Adaptive difficulty

In WebXR: Babylon.js provides **navigation mesh** (navmesh) and crowd agents for autonomous NPC pathfinding.

Note: Not all immersive applications need AI — it depends on your design goals.

Navmesh: https://doc.babylonjs.com/features/featuresDeepDive/crowdNavigation/createNavMesh/

Component Roles in EDIS

Experience Dementia in Singapore

What components do you need here:

- Rendering — displays the 360° kitchen environment
- Input Handler — gaze-based navigation between hotspots
- Audio — plays ambient sounds and caregiver narration
- Physics — is there any?
- AI — Scripted linear experience

Source: https://dementia.org.sg/2022/03/30/experience-dementia-in-singapore/

Designing Your Architecture

Don’t just use every component—design your unique mix:

1. Understand Your Requirements

- What experience are you creating? (game, training, visualization, social)
- What interactions does your design demand?
- What are your performance constraints?

2. Research Available Options

- Explore existing frameworks, libraries, and patterns
- Learn from similar applications—what did they use?
- Watch tutorials, read documentation, study code samples

3. Curate Your Architecture

- Select only the components that serve your design
- Avoid “kitchen sink” syndrome—every system adds complexity
- Create custom components when existing ones don’t fit

Entity Component System (ECS)

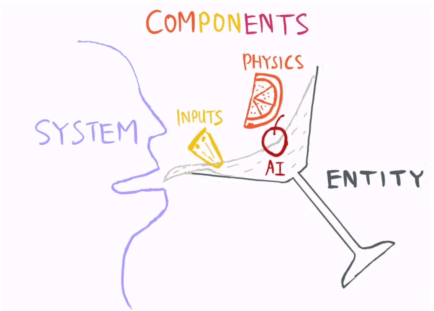
Modern simulations/games often use **ECS architecture**

Traditional OOP approach:

- Objects inherit from base classes
- Tight coupling between behaviors
- Hard to add new behaviors dynamically

ECS approach:

- **Entities** — unique IDs (player, enemy, light)
- **Components** — data containers (position, mesh, physics)
- **Systems** — logic processors (rendering, physics, AI)



Video segment: https://youtu.be/OKD4jrnn4WE?t=1839

Why ECS Over OOP?

1. **Flexibility**
   - Add/remove behaviors at runtime
   - No rigid inheritance hierarchies
2. **Performance**
   - Data-oriented design enables cache-friendly processing
   - Systems process components in batches
3. **Maintainability**
   - Clear separation of data and logic
   - Easier to test and debug
   - Teams work on different systems independently

Top reason: Composition enables flexible, reusable, performant code

A-Frame: ECS in Practice

A-Frame is a WebXR framework built on top of Three.js that implements ECS

Entities (HTML elements):

`<a-entity>` — empty container, just an ID

Components (HTML attributes):

```
position="1 2 3" geometry="primitive: box" material="color: red"
```

Systems (JavaScript):

```
AFRAME.registerSystem('game', {init: fn, tick: fn});
```

Key insight: HTML attributes = Components, making ECS intuitive for web developers

A-Frame: Defining and Using Components

Defining a custom component (JavaScript):

```
AFRAME.registerComponent('spin', {
schema: { speed: { default: 1 } },
tick: function (time, delta) {
    this.el.object3D.rotation.y += this.data.speed * delta / 1000;
  }
});
```

Attaching components to an entity (HTML):

```
<a-entity
geometry="primitive: box"
material="color: red"
position="0 1.5 -3"
spin="speed: 2">
</a-entity>
```

Key: Components are data (attributes), entities are containers, systems process them.

Snap Lens Studio: AR Development Platform

Architecture: Visual editor + TypeScript scripting engine

- Component-based: Scripts attach to Scene Objects
- Event-driven lifecycle (onAwake, onUpdate, onDestroy)
- Deploys to Snapchat app and Spectacles AR glasses

TypeScript Example:

```
@component
export class TapHandler extends BaseScriptComponent {
  @input sceneObject: SceneObject;
    onAwake() {
      this.createEvent('TapEvent').bind(() => this.onTap());
    }
    onTap() {
      this.sceneObject.enabled = !this.sceneObject.enabled;
    }
}
```

Source: AWE Asia 2026, Snap Engineer Atley Loughridge

Babylon.js: Bring Your Own Architecture

Babylon.js provides powerful tools but **no enforced architecture**

What Babylon.js provides:

- Scene graph with meshes, cameras, lights
- Component-like behaviors (attachable to meshes)
- Observables for event handling, WebXR session management

What developers must design:

- How to structure logic (classes, modules, ECS?)
- How to manage state and separate concerns

Result: More flexibility, but requires conscious architectural decisions

WebXR Architecture: A-Frame vs Babylon.js

A-Frame (ECS built-in)	Babylon.js (BYO Architecture)
Declarative HTML approach	Programmatic JavaScript approach
ECS architecture enforced	Architecture up to developer
Easier for beginners	More control for experts
Three.js under the hood	Direct WebGL/WebGPU rendering
Smaller community, fewer resources	Larger community, extensive docs
Good for rapid prototyping	Good for complex applications

Bottom line: A-Frame gives you structure; Babylon.js gives you freedom — choose based on team experience and project needs.

WebXR Application Structure

Typical WebXR app structure:

1. **Scene Graph**
   - Hierarchical tree of objects, parent-child transformations
2. **Rendering Loop**
   - `requestAnimationFrame` or `XRSession.requestAnimationFrame`
   - Update logic → render frame (60–90+ FPS)
3. **Component System**
   - Meshes, materials, lights, cameras, physics bodies
   - Custom behaviors and scripts attached to objects

XR Rendering: Time Budget

Why frame timing matters in XR:

Target FPS	Time per frame	Typical use
60 FPS	16.7ms	Desktop VR, Mobile AR
72 FPS	13.9ms	Quest default mode
120 FPS	8.3ms	High-end PC VR

Each frame must complete:

- Input processing (controller, hand tracking)
- Physics simulation and collision detection
- Game logic / AI updates
- **Stereo rendering** (render twice — left & right eye!)

Miss the deadline → dropped frames → motion sickness!

From Software to Hardware

Now that we understand software architecture, let's explore the hardware that runs these immersive applications.

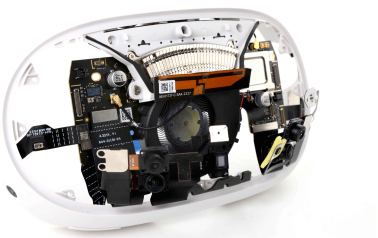
Key question: What physical components make XR possible?

- What's inside a VR headset?
- How do lenses create the VR illusion?
- What different types of AR devices exist?

Inside an HMD: Meta Quest 3

Key Hardware Components:

- Dual LCD displays (2064 × 2208 per eye, 120Hz)
- Pancake lenses — thinner/clearer than fresnel lenses
- Touch Plus controllers
- Motion tracking sensors
- Cameras (inside-out tracking, passthrough, depth sensor)
- Snapdragon XR2 Gen 2, 8GB RAM



Source: iFixit Meta Quest 3 Teardown — ifixit.com/News/84572

Smartphone vs HMD Hardware

Similarities:

- Display screen, CPU/GPU, Motion sensors (IMU), Cameras, Battery

Key Difference: Magnifying Lenses

- HMDs use lenses to magnify the close display
- Creates wide field of view, enables virtual image at optical infinity
- Smartphone cardboard viewers add lenses to phones

Without lenses, you'd just see a tiny screen up close — lenses create immersion.

AR Hardware Landscape

Augmented Reality devices come in various form factors:

1. **AR Glasses** — lightweight, everyday wearable
2. **AR Headsets** — higher capability, professional use
3. **Mobile AR** — smartphone/tablet cameras

Let’s explore recent AR hardware across these categories

AR Glasses

Meta Ray-Ban Smart Glasses

- Form: Consumer glasses with integrated electronics (49g)
- Features: Cameras, speakers, mic, AI
- Display: *No display*

Meta Ray-Ban Meta display

- Form: Consumer glassess with micro-display
- Features: Same as before but with EMG input
- Display: Micro-LED monocular display

Xreal Air / Air 2 Pro

- Form: Consumer glasses with integrated virtual display (tethered)
- Features: 130” screen, USB-C
- Display: 1080p OLED, 46–49° FOV

Sources: meta.com, xreal.com, ray-ban.com

AR H/W Example: Snap Spectacles (5th Gen)

Positioned between smart glasses and full AR headsets

- Form: Standalone untethered glasses (226g)
- Display: 46° FOV stereo waveguide, LCoS projectors
- Compute: Dual Snapdragon processors, 45 min battery
- Input: Hand tracking, voice, 6DoF inside-out tracking
- 37 PPD, 13ms motion-to-photon latency

Takeaway: True stereo AR in glasses form factor — sits between smart glasses (Meta Ray-Ban) and full-blown AR headsets (HoloLens).

Source: spectacles.com/explore

AR vs VR Hardware: Key Differences

Component	AR (e.g., HoloLens, Spectacles)	VR (e.g., Quest 3)
Display Tech	Waveguide/combiner (see-through)	LCD/OLED panels (opaque)
Optics	Diffractive waveguide, birdbath	Fresnel or pancake lenses
FOV	40–70° (limited by waveguide)	90–120° (larger panels possible)
Processor	Mobile SoC (power constrained)	XR-optimized SoC (more headroom)
Cameras	RGB + depth for world understanding	Tracking + passthrough + depth
Depth Sensing	Often ToF or structured light	ToF or stereo depth
Control Input	Hand tracking, gaze, voice	Controllers, hand tracking
Weight Target	<300g (glasses form factor goal)	400–600g (headset acceptable)

Key insight: AR requires see-through optics (harder, narrower FOV) while VR uses simpler opaque displays. AR prioritizes lightweight form; VR prioritizes immersion.

AR Headsets

Microsoft HoloLens 2

- Form: Optical see-through AR headset
- Features: Hand tracking, enterprise focus
- Display: 52° FOV, waveguide, standalone

Magic Leap 2

- Form: Optical see-through AR headset
- Features: Eye tracking, enterprise/medical
- Display: 70° FOV, waveguide with dimming

Apple Vision Pro

- Form: Video passthrough spatial computer
- Features: Eye + hand tracking
- Display: 23M pixels, micro-OLED

Sources: microsoft.com, magicleap.com, apple.com

AR Hardware: Key Differences

Form Factor Trade-offs:

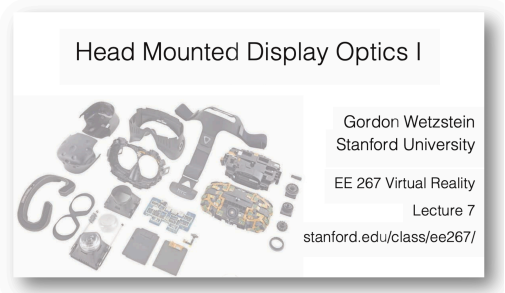
- **Glasses** — lightweight, socially acceptable, limited capability
- **Headsets** — more powerful, wider FOV, bulkier
- **Mobile** — ubiquitous, no extra hardware, limited tracking

Display Technology:

- **Optical see-through** — direct view + overlay (HoloLens, Magic Leap)
- **Video passthrough** — cameras + digital merge (Vision Pro, Quest 3)
- **Virtual display** — screen projection only (Xreal, Ray-Ban)

Use Cases: Enterprise (training, remote assist), Consumer (entertainment), Medical/Industrial

Connecting Hardware to Software



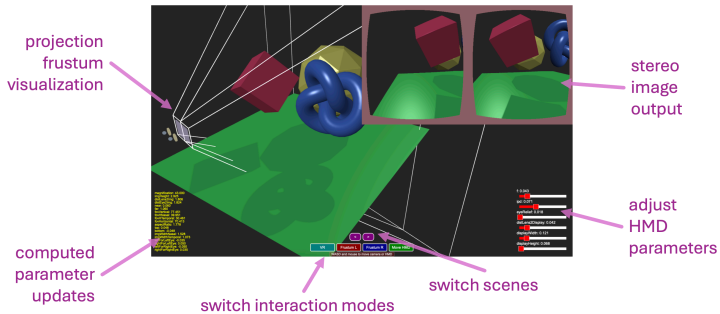
Stanford EE267: `https://stanford.edu/class/ee267/lectures/lecture7.pdf`

Immersification Video: `https://youtu.be/OKD4jrnn4WE`

HMD Simulator

Purpose: Understanding Hardware-Software Connection

- Visualize how HMD hardware parameters affect rendering
- Experiment with lens properties, IPD, FOV, eye relief
- See real-time impact on the rendered view



Summary

Today we covered:

- Software components (Rendering, Physics, Input, Audio, AI)
- ECS architecture and its advantages over OOP
- A-Frame vs Babylon.js architectural approaches
- HMD hardware components (Quest 2 example)
- AR hardware landscape (glasses, headsets, mobile)

Next: Week06 — Deep dive into HMD optics theory