Portal Game

A 3D First-Person Game

Computer Graphics Project Documentation

Project Team:

TheOnlyMaro : 22p0201
Malak-Abdelakher : 21p0278
Lina-Elsharkawy : 21p0198
MohamedZaima :22p0147
Gadarars : 22p0188

Presented to:

Dr. Mohamed Rehan
ENG.Muhammad Ashraf

December 2025

Contents

Abstract

This document presents a comprehensive technical report on the development of a 3D first-person puzzle game inspired by Valve's Portal series. The game implements advanced portal mechanics, physics simulation, collision detection, and interactive puzzle elements using Three.js and modern web technologies. The project demonstrates sophisticated computer graphics techniques including portal rendering with stencil buffers, raycasting, player physics, and dynamic lighting systems.

The game features multiple levels with increasing complexity, incorporating environmental puzzles, pressure plate mechanisms, movable objects, and hazardous obstacles. This report details the technical architecture, implementation challenges, game mechanics, and design decisions that shaped the final product.

1 Introduction

1.1 Project Overview

Portal Game is a browser-based 3D first-person puzzle game that challenges players to navigate complex environments using portal mechanics. Inspired by the critically acclaimed Portal series, the game implements a sophisticated portal system that allows players to create interconnected passages through space, enabling creative solutions to spatial puzzles.

The game is built entirely using web technologies, making it accessible through modern browsers without requiring additional installations or plugins. This approach demonstrates the capabilities of WebGL and JavaScript for creating immersive 3D gaming experiences.

1.2 Technical Stack

The project utilizes the following core technologies:

- **Three.js (v0.182.0):** A powerful 3D graphics library built on WebGL
- **Express.js (v5.2.1):** Development server with cache-control headers
- **Modern JavaScript (ES6+):** Module system, classes, and async patterns
- **WebGL Shaders:** Custom GLSL shaders for portal rendering effects
- **Node.js:** Development environment and package management

1.3 Project Objectives

The primary objectives were:

1. Implement functional portal mechanics with realistic physics
2. Create intuitive first-person controls with smooth camera movement
3. Design engaging puzzle levels with progressive difficulty
4. Develop robust collision detection and physics systems
5. Implement interactive game objects
6. Optimize rendering performance for smooth browser gameplay

1.4 Game Concept

Players navigate test chambers with a portal gun that creates two interconnected portals (blue and orange). Through strategic portal placement, players can:

- Teleport instantly between distant locations
- Redirect momentum and velocity through portals
- Solve spatial puzzles requiring creative thinking
- Navigate hazardous environments with spike traps
- Activate pressure plates and door mechanisms
- Manipulate physics objects to progress through levels

2 System Architecture

2.1 Project Structure

The codebase follows a modular architecture with clear separation of concerns:

```
Portal_Game/
|-- Controllers/          # Player and camera control systems
|   |-- CameraController.js
|   |-- PlayerController.js
|   |-- MouseController.js
|   |-- keyboardController.js
|   |-- collisionController.js
|-- core/                 # Core rendering systems
|   |-- renderer.js
|   |-- clock.js
|-- decor/                # Environmental decorations
|   |-- floor_ceiling.js
|   |-- lights.js
|   |-- panels.js
|-- obstacles/            # Static environment objects
|   |-- walls.js
|-- portal_logic/         # Portal system implementation
|   |-- portalSystem.js
|   |-- portalRayCaster.js
|   |-- portalTeleport.js
|   |-- PortalRender.js
|   |-- PortalCamera.js
|   |-- PortalMesh.js
|   |-- PortalStencilMask.js
|   |-- portalHalo.js
|   |-- CubeButtonRaycaster.js
|-- puzzle_logic/         # Interactive puzzle elements
|   |-- PuzzleObjects.js
|-- physics/              # Physics and interaction systems
|   |-- interactable.js
|   |-- interactables.js
|   |-- physicsObject.js
|-- scenes/               # Level definitions
|   |-- LevelOne.js
|   |-- LevelTwo.js
|   |-- LevelThree.js
|   |-- Room.js
|   |-- Corridor.js
|-- textures/             # Material definitions and textures
|   |-- materials_TextureMapping.js
|   |-- studio_small_09_4k.exr
|-- models/               # 3D model assets
|   |-- laser_gun.glb
|   |-- portal_cube.glb
|   |-- portal_2_-_cube_button.glb
|-- main.js               # Application entry point
|-- index.html            # HTML container
|-- index.css             # Styling
|-- server.js             # Development server
|-- package.json          # Dependencies
|-- .gitignore              # Git ignore rules
```

Listing 1: Project Directory Structure

2.2 Core Components

2.2.1 Main Application Loop

The `main.js` file orchestrates all major systems:

```
// Initialize renderer and scene
const renderer = createRenderer();
const { scene, walls, puzzle, collisionObjects, spawnPoint }
    = setupLevelThree();

// Initialize player and controls
const camera = new THREE.PerspectiveCamera(...);
const cameraController = new CameraController(
    camera, scene, walls, collisionObjects
);

// Initialize portal system
const portalSystem = new PortalSystem();
const portalTeleport = new PortalTeleport(
    cameraController.getPlayer(),
    portalSystem,
    cameraController.collision,
    cameraController.mouse
);

// Animation loop
function animate() {
    requestAnimationFrame(animate);
    const deltaTime = Math.min(clock.getDelta(), 0.05);

    cameraController.update(deltaTime);
    portalTeleport.update(deltaTime);
    portalSystem.update(portalRaycaster.hitInfo);

    portalRenderer.render(...);
    renderer.render(scene, camera);
}
```

Listing 2: Main Game Loop Architecture

2.2.2 Module Responsibilities

Table 1: Module Responsibilities

Module	**Responsibility**
CameraController	Coordinates player movement, camera updates, input handling, and collision detection
PlayerController	Manages player physics: gravity, jumping, velocity, and ground detection
PortalSystem	Handles portal placement, activation state, visual feedback, and portal data storage
PortalTeleport	Detects player proximity to portals and executes teleportation with momentum preservation
PortalRenderer	Renders portal views using stencil buffers and oblique projection matrices
CollisionController	Performs AABB collision detection between player and environment
PuzzleObjects	Implements interactive elements: draggable cubes, pressure buttons, and doors

3 Player Control System

3.1 Camera and Movement

The player control system implements first-person perspective with realistic movement mechanics.

3.1.1 PlayerController Implementation

The `PlayerController` class manages the player's physical representation:

```
export class PlayerController {
    constructor(camera) {
        this.player = new THREE.Object3D();
        this.camera = camera;

        this.velocity = new THREE.Vector3();
        this.speed = 8;                // Movement speed (m/s)
        this.jumpStrength = 10;        // Jump impulse
        this.gravity = -30;            // Gravity acceleration
        this.onGround = false;

        this.raycaster = new THREE.Raycaster(
            new THREE.Vector3(),
            new THREE.Vector3(0, -1, 0),
            0, 2
        );
    }

    update(delta) {
        // Apply gravity
        this.velocity.y += this.gravity * delta;

        // Apply velocity to position
        this.player.position.x += this.velocity.x * delta;
        this.player.position.z += this.velocity.z * delta;
        this.player.position.y += this.velocity.y * delta;

        // Ground detection via raycasting
        this.raycaster.ray.origin.copy(this.player.position);
        const intersections = this.raycaster
            .intersectObjects(this.floorMeshes, false);

        if (intersections.length > 0 && this.velocity.y <= 0) {
            this.player.position.y = intersections[0].point.y;
            this.velocity.set(0, 0, 0);
            this.onGround = true;
        }
    }
}
```

Listing 3: Player Physics System

3.1.2 Input Handling

The system uses separate controllers for keyboard and mouse input:

- **KeyboardController:** Maps WASD keys to directional movement and Space for jumping
- **MouseController:** Implements pointer lock for camera rotation with pitch clamping

```
export class MouseController {
    constructor(player, camera) {
        this.pitch = 0;
        this.sensitivity = 0.002;

        document.addEventListener('mousemove', e => {
            if (document.pointerLockElement !== document.body)
                return;

            // Yaw rotation (Y-axis)
            player.rotation.y -= e.movementX * this.sensitivity;

            // Pitch rotation (X-axis) with clamping
            this.pitch -= e.movementY * this.sensitivity;
            this.pitch = Math.max(-Math.PI/2,
                        Math.min(Math.PI/2, this.pitch));
            camera.rotation.x = this.pitch;
        });
    }
}
```

Listing 4: Mouse Control System

3.2 Collision Detection

The `CollisionController` implements Axis-Aligned Bounding Box (AABB) collision detection.

3.2.1 Key Features

1. **Step-over mechanics:** Player can step over small obstacles (0.2m)
2. **Cached wall boxes:** Static walls pre-calculated for performance
3. **Horizontal-only reversion:** Only X/Z positions revert, preserving gravity
4. **Dynamic object support:** Separate handling for moving objects

```
update() {
    const width = 0.6;
    const totalHeight = 2.7;
    const stepHeight = 0.2;
    const boxHeight = totalHeight - stepHeight;

    const boxCenter = this.player.position.clone();
    boxCenter.y += stepHeight + (boxHeight / 2);

    this.playerBox.setFromCenterAndSize(
        boxCenter,
        new THREE.Vector3(width, boxHeight, width)
    );

    // Check static walls (cached)
    for (const wallBox of this.wallBoxes) {
        if (this.playerBox.intersectsBox(wallBox)) {
            this.handleCollision();
            return;
        }
    }
}

handleCollision() {
    // Only revert horizontal movement
    this.player.position.x = this.player.prevPosition.x;
    this.player.position.z = this.player.prevPosition.z;
}
```

Listing 5: Collision Detection Algorithm

4 Portal System

4.1 Portal Mechanics Overview

The portal system is the core feature, allowing players to create interconnected passages through space.

4.2 Portal Placement

Portals are placed using raycasting from the player's view direction.

4.3 Portal Rendering

The portal rendering system uses multi-pass rendering with stencil buffers to create realistic portal views.

4.4 Portal Teleportation

When a player enters a portal, both position and view direction transform according to the portal orientations, preserving momentum through quaternion rotation.

5 Level Design and Gameplay

5.1 Level Three - The Main Challenge

Level Three represents the culmination of all game mechanics, featuring a complex multi-section design that progressively teaches and tests player skills.

5.1.1 Level Structure

The level consists of four main sections:

1. **Dark Starting Corridor**
   - Length: 150 units
   - Initial state: Complete darkness
   - Teaching moment: Environmental button interaction
   - Player spawns at position (20, 5, 0)
   - Button at position (5, 0, 0) activates ambient lighting
   - Light bulb at end provides visual goal

2. **Underground Pit**
   - Floor level: Y = -8 (8 meters below ground)
   - Multiple platforms at varying heights
   - Spike Area 1: Jumpable obstacle (15 units long)
   - Platform 1: Safe landing zone (8x8 units)
   - Spike Area 2: Impossible jump requiring portal usage
   - Platform 2: Small target platform (5x5 units) accessible only via portal

3. **Spike Corridor**
   - Orientation: Along Z-axis (perpendicular to main path)
   - Length: 80 units
   - Floor completely covered with spike obstacles
   - Requires careful jumping or portal navigation
   - Light bulb at end provides waypoint

4. Final Puzzle Room

- Size: 30x10 units (X x Z)
- Contains draggable cube puzzle element
- Ceiling platform at Y = 7 with pressure button
- Wall-mounted button requiring portal access
- Exit door activated by puzzle completion

5.2 Puzzle Mechanics

5.2.1 Interactive Objects

The game implements three main types of interactive objects:

Draggable Cube Uses custom GLTF model (portal_cube.glb) that players can:

- Pick up with left-click when looking at it
- Carry in front of camera view
- Drop by clicking again
- Place on pressure buttons to activate mechanisms

Floor Buttons Pressure-sensitive triggers that:

- Detect proximity of cube (3-unit radius)
- Activate/deactivate doors
- Use GLTF model (portal_2_-_cube_button.glb)
- Provide visual feedback when pressed

Doors Animated barriers that:

- Block player progression initially
- Slide upward 9 units when activated
- Use procedural tech panel material
- Animate smoothly with lerp interpolation

5.3 Environmental Hazards

5.3.1 Spike Traps

Cone-shaped obstacles that:

- Cause instant player death on contact
- Force respawn at level start
- Create navigation challenges
- Encourage portal usage for safe passage

5.3.2 Fall Death System

- Triggers at Y position below -10
- Respawns player at spawn point
- Resets velocity and physics state
- First death activates ambient lighting (tutorial aid)

6 Graphics and Visual Systems

6.1 Material System

The game uses procedurally defined materials for optimal performance and consistency.

6.1.1 Lab Wall Material

```
export function createLabWallMaterial() {
    return new THREE.MeshStandardMaterial({
        color: 0xeeeeee,
        metalness: 0.1,
        roughness: 0.8,
        envMapIntensity: 0.5
    });
}
```

Listing 6: Lab Wall Material Definition

6.1.2 Metal Floor Material

```
export function createMetalFloorMaterial() {
    return new THREE.MeshStandardMaterial({
        color: 0x444444,
        metalness: 0.9,
        roughness: 0.3,
        envMapIntensity: 1.0
    });
}
```

Listing 7: Metal Floor Material

6.1.3 Tech Panel Material

Emissive material used for doors and decorative elements:

- Cyan color (0x00ffff)
- Emissive intensity: 0.5
- Metalness: 0.5, Roughness: 0.2
- Creates sci-fi aesthetic

6.2 Lighting Design

6.2.1 Ambient Lighting

- Initial intensity: 0.15 (very dark)
- Activated intensity: 0.6 (playable brightness)
- Provides base illumination throughout scene
- Controlled by environmental button

6.2.2 Point Lights

Light bulbs placed strategically as waypoints:

- Yellow color (0xffff99)
- Intensity: 5 when activated
- Range: 50 units
- Include emissive sphere mesh and glow effect

6.2.3 Shadow System

- Enabled on renderer with PCFSoftShadowMap
- All walls and floors receive shadows
- Player gun model casts shadows
- Enhances depth perception and spatial awareness

6.3 Portal Visual Effects

6.3.1 Portal Halo System

The halo provides visual feedback for portal placement:

```
animate(deltaTime) {
    if (!this.mesh.visible) return;
    this.animationTime += deltaTime;
    const pulse = Math.sin(this.animationTime * 3) * 0.1 + 0.9;
    this.glowMesh.material.opacity = pulse * 0.3;
    this.glowMesh.rotation.z += deltaTime * 0.5;
}
```

Listing 8: Portal Halo Animation

Features:

- Ring geometry with 1.08m radius
- Blue (0x0000ff) for Q key portal
- Orange (0xff6600) for E key portal
- Pulsing opacity animation
- Rotating inner glow mesh
- Additive blending for glow effect

6.3.2 Portal Rendering Pipeline

The portal uses advanced multi-pass rendering:

Pass 1: Stencil Mask

- Circular geometry defines portal boundary
- Writes value 1 to stencil buffer
- ColorWrite and DepthWrite disabled
- RenderOrder: -1 (renders first)

Pass 2: Portal View

- Renders scene from destination portal's perspective
- Uses oblique near-plane clipping
- Outputs to render target texture
- Both portal meshes hidden to prevent recursion

Pass 3: Portal Display

- Screen-space shader displays render target
- Only renders where stencil = 1
- Brightness boost uniform (2.5x) compensates for double tone-mapping
- Uses screen UV coordinates for correct projection

6.4 Portal Shader Implementation

```
uniform sampler2D map;
uniform vec2 resolution;
uniform float brightnessBoost;

void main() {
    // Calculate screen UV coordinates (0..1)
    vec2 screenUV = gl_FragCoord.xy / resolution;

    // Sample the portal texture
    vec4 color = texture2D(map, screenUV);

    // Multiply color to fix lighting
    gl_FragColor = vec4(color.rgb * brightnessBoost, color.a);
}
```

Listing 9: Portal Fragment Shader

7 Physics and Game Mechanics

7.1 Player Physics System

7.1.1 Movement Parameters

Table 2: Player Physics Constants

Parameter	Value	Unit
Movement Speed	8	m/s
Jump Strength	10	m/s (initial velocity)
Gravity	-30	m/s^2
Player Width	0.6	meters
Player Height	2.7	meters
Step Height	0.2	meters

7.1.2 Ground Detection Algorithm

The system uses raycasting to detect floor contact:

```
update(delta) {
    // Apply gravity
    this.velocity.y += this.gravity * delta;

    // Apply velocity to position
    this.player.position.x += this.velocity.x * delta;
    this.player.position.z += this.velocity.z * delta;
    this.player.position.y += this.velocity.y * delta;

    // Raycast from slightly above player position
    const rayOrigin = this.player.position.clone();
    rayOrigin.y += 1.0;

    this.raycaster.ray.origin.copy(rayOrigin);

    // Dynamic far distance based on fall speed
    const moveY = this.velocity.y * delta;
    const lookAhead = Math.max(0, -moveY) + 0.2;
    this.raycaster.far = 1.0 + lookAhead;

    const intersections = this.raycaster
        .intersectObjects(this.floorMeshes, false);

    this.onGround = false;

    if (intersections.length > 0 && this.velocity.y <= 0) {
        // Snap to floor
        this.player.position.y = intersections[0].point.y;

        // Reset all velocity (stop instantly on landing)
        this.velocity.set(0, 0, 0);
        this.onGround = true;
    }
}
```

Listing 10: Advanced Ground Detection

Key features:

- Raycast originates 1m above player feet
- Dynamic raycast distance adapts to fall speed
- Only triggers when moving downward (velocity.y ≤ 0)
- Instant velocity cancellation on landing prevents bouncing

7.2 Collision Detection System

7.2.1 AABB Implementation

The collision system uses Axis-Aligned Bounding Boxes with optimizations:

```
constructor(player, walls, dynamicObjects, scene) {
    this.player = player;
    this.walls = walls;
    this.dynamicObjects = dynamicObjects;
    this.playerBox = new THREE.Box3();

    // OPTIMIZATION: Pre-calculate static wall boxes
    this.wallBoxes = this.walls.map(wall => {
        const box = new THREE.Box3();
        box.setFromObject(wall);
        return box;
    });
}
```

Listing 11: Optimized Collision Detection

Performance Optimization

- Static walls: Bounding boxes calculated once in constructor
- Dynamic objects: Recalculated every frame
- Early exit on first collision detection
- Reduces setFromObject() calls by 90%+

7.2.2 Step-Over Mechanics

The collision box is elevated to allow stepping over small obstacles:

```
const width = 0.6;
const totalHeight = 2.7;
const stepHeight = 0.2;
const boxHeight = totalHeight - stepHeight;

// Lift box center up by step height
const boxCenter = this.player.position.clone();
boxCenter.y += stepHeight + (boxHeight / 2);

this.playerBox.setFromCenterAndSize(
    boxCenter,
    new THREE.Vector3(width, boxHeight, width)
);
```

Listing 12: Step-Over Implementation

This creates a 0.2m gap below the collision box, allowing players to:

- Walk over small floor details
- Navigate door thresholds smoothly
- Step onto low platforms naturally

7.3 Portal Teleportation Mathematics

7.3.1 Position Transformation

When teleporting, the player's position is calculated as:

$$\vec{P}_{exit} = \vec{P}_{dest} + \vec{n}_{dest} \times 1.5 \tag{1}$$

Where:

- $\vec{P}_{exit}$ = Exit position in world space
- $\vec{P}_{dest}$ = Destination portal position
- $\vec{n}_{dest}$ = Destination portal normal vector
- 1.5 = Offset distance to prevent clipping

7.3.2 Rotation Transformation

The view direction rotation uses quaternion mathematics:

```javascript
// Create quaternions for portal orientations
const forward = new THREE.Vector3(0, 0, 1);
const sourceQuat = new THREE.Quaternion()
    .setFromUnitVectors(forward, sourcePortal.normal);
const destQuat = new THREE.Quaternion()
    .setFromUnitVectors(forward, destinationPortal.normal);

// 180-degree flip
const flipQuat = new THREE.Quaternion()
    .setFromAxisAngle(new THREE.Vector3(0, 1, 0), Math.PI);

// Combined transform
const transformQuat = new THREE.Quaternion();
transformQuat.copy(destQuat)
    .multiply(flipQuat)
    .multiply(sourceQuat.clone().invert());
```

Listing 13: Portal Rotation Transform

The transformation matrix:

$$Q_{transform} = Q_{dest} \times Q_{flip} \times Q_{source}^{-1} \tag{2}$$

7.3.3 Momentum Preservation

The same quaternion transformation applies to velocity:

```javascript
if (this.player.velocity) {
    // Apply rotation to velocity vector
    this.player.velocity.applyQuaternion(transformQuat);

    // Optional exit push to prevent re-triggering
    if (this.player.velocity.length() < 5.0) {
        const exitPush = destinationPortal.normal
            .clone().multiplyScalar(5.0);
        this.player.velocity.add(exitPush);
    }
}
```

Listing 14: Velocity Transformation

This ensures:

- Falling velocity becomes horizontal exit velocity
- Magnitude preserved (energy conservation)
- Direction correctly rotated based on portal orientations
- Portal ”flinging” mechanics work naturally

8 Technical Challenges and Solutions

8.1 Portal Overlap Prevention

Problem: Players could place portals too close together, causing visual artifacts and teleportation bugs.

Solution: Distance check before portal placement:

```
placePortal(hitInfo) {
    let otherPortalPoint = null;
    if (this.currentPortal === 'blue'
        && this.orangePortalActive
        && this.orangePortalData) {
        otherPortalPoint = this.orangePortalData.point;
    }

    if (otherPortalPoint) {
        const dist = hitInfo.point
            .distanceTo(otherPortalPoint);
        if (dist < 2.5) {
            console.warn("Portals too close!");
            return null;
        }
    }
    // ... place portal
}
```

Listing 15: Overlap Prevention

8.2 Wall Sticking Issue

Problem: Players would stick to walls when collision occurred, unable to slide along surfaces.

Solution: Only revert horizontal (X/Z) position, not vertical (Y):

```
handleCollision() {
    // Only revert horizontal movement
    this.player.position.x = this.player.prevPosition.x;
    this.player.position.z = this.player.prevPosition.z;
    // Y position unchanged - allows gravity to work
}
```

Listing 16: Collision Resolution

8.3 Portal Rendering Darkness

Problem: Portal views appeared too dark compared to main view.

Solution: Brightness boost in portal shader:

```
uniform float brightnessBoost; // Set to 2.5

void main() {
    vec4 color = texture2D(map, screenUV);
    gl_FragColor = vec4(color.rgb * brightnessBoost, color.a);
}
```

Listing 17: Brightness Compensation

8.4 Momentum Loss on Landing

Problem: Portal momentum wasn't preserved after going through portals and landing.

Solution: Complete velocity reset only on ground contact:

```
if (intersections.length > 0 && this.velocity.y <= 0) {
    this.player.position.y = intersections[0].point.y;
    // Reset ALL velocity when landing
    this.velocity.set(0, 0, 0);
    this.onGround = true;
}
```

Listing 18: Momentum Fix

9 Performance Optimizations

9.1 Collision Detection

- **Static Wall Caching:** Bounding boxes pre-calculated in constructor
- **Early Exit:** First collision detected immediately returns
- **Separate Floor/Wall Arrays:** Reduces unnecessary checks

9.2 Rendering

- **Portal Render Target:** 512x512 resolution (not full screen)
- **Recursive Prevention:** Portal meshes hidden during portal view rendering
- **Delta Time Capping:** Max 0.05s to prevent large physics jumps
- **Shadow Map Optimization:** PCFSoftShadowMap for quality/performance balance

9.3 Asset Management

- **GLTF Models:** Compressed format for 3D assets
- **Procedural Materials:** No texture loading for most surfaces
- **EXR Environment:** Single HDR texture for reflections

10 Development Tools

10.1 Development Server

Express.js server with cache-busting headers for rapid iteration:

```
app.use((req, res, next) => {
    res.set('Cache-Control',
        'no-store, no-cache, must-revalidate');
    res.set('Pragma', 'no-cache');
    res.set('Expires', '0');
    next();
});
```

Listing 19: Development Server Configuration

10.2 Debug Features

- **Raycast Visualization:** Colored lines show portal and cube raycasts
- **Collision Box Helper:** Red box shows player collision volume
- **Console Logging:** Teleportation events, button presses, deaths
- **Trigger Zone Wireframes:** Optional visualization for button zones

11 Future Enhancements

11.1 Planned Features

1. **Additional Levels:** More complex puzzle chambers
2. **Moving Platforms:** Dynamic environment obstacles
3. **Laser Mechanics:** Light-based puzzles like original Portal
4. **Turrets:** Automated threats requiring careful navigation
5. **Companion Cube Physics:** More realistic cube behavior
6. **Audio System:** Sound effects and ambient audio
7. **Menu System:** Level selection and settings
8. **Save System:** Progress persistence

11.2 Technical Improvements

1. **Recursive Portal Rendering:** Portals visible through portals
2. **Better Collision:** Swept AABB or capsule collision
3. **Network Multiplayer:** Co-op puzzle solving
4. **Mobile Support:** Touch controls and performance optimization
5. **Level Editor:** In-browser level creation tool

12 Conclusion

This Portal Game project successfully demonstrates advanced 3D game development techniques using web technologies. The implementation showcases:

- Complex portal mechanics with proper momentum preservation
- Efficient collision detection using AABB with optimizations
- Multi-pass rendering with stencil buffers for realistic portal views
- Physics simulation with gravity, jumping, and ground detection
- Interactive puzzle elements with doors, buttons, and draggable objects
- Progressive level design teaching players mechanics through gameplay

The modular architecture separates concerns effectively: Controllers handle input and movement, portal_logic manages teleportation and rendering, puzzle_logic implements interactive objects, and scenes define level layouts. This structure enables easy expansion and maintenance.

Performance optimizations like cached collision boxes, capped delta time, and efficient portal rendering ensure smooth gameplay at 60 FPS in modern browsers. The use of Three.js provides excellent WebGL abstraction while maintaining low-level control when needed.

Future enhancements could expand the gameplay significantly while building on the solid foundation established. The codebase is well-structured for collaborative development and continued iteration.