Université Paris Saclay

Virtual and Mixed Reality

Master 2 in Mechatronics, Machine Vision, Artificial Intelligence

---

Lab session

Using and Programming Virtual Reality Headsets

Student:
Yann Terrom
Dat Nguyen
Giuseppe Ricciardi
Chakib Gamir

Supervisor:
Jean-Yves Didier

Master:
M2MMVAI

Academic Year
2023-2024

Contents

List of Figures

Listings

1 Introduction

This report provides an overview of the extended reality (XR) game development project undertaken by the team consisting of Yann Terrom, Dat Nguyen, Giuseppe Ricciardi, and Chakib Gamir. The primary objective of the project was to design and implement a 3D game for virtual reality headsets, such as Oculus, focusing on the classic puzzle known as the Tower of Hanoi.

1.1 Team Members

- Yann Terrom - Contributed to the project's structural development, including setting up the project using npm. Worked on creating fundamental game objects such as disks, the main game structure, background music, and the 3D environment.
- Dat Nguyen - Took a hands-on approach in the implementation of the game logic, contributing to the aspects of the gaming experience. Worked on shaping the behavior, rules, and interactive elements within the game.
- Giuseppe Ricciardi - Collaborated on object selection and played a crucial role in refining the physics of the game, incorporating elements like gravity and collision detection.
- Chakib Gamir - Contributed to the project's structural development, including setting up the project using npm. Worked on creating fundamental game objects such as disks, the main game structure, background music, and the 3D environment.

After establishing individual contributions, the team members collaborated to develop the game logic, including defining the rules and overall gameplay.

1.2 Tower of Hanoi Rules

The Tower of Hanoi is a mathematical puzzle that involves three pegs and a number of disks of different sizes. The puzzle starts with the disks in a neat stack in ascending order of size on one peg, the smallest at the top. The objective is to move the entire stack to another peg, obeying the following rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty peg.
3. No disk may be placed on top of a smaller disk.

The game is typically played with three pegs, but the number of pegs can vary. The minimal number of moves required to solve a Tower of Hanoi puzzle with n disks is $2^n - 1$.

1.3 Technical Framework

The Tower of Hanoi extended reality game was developed with a focus on immersive experiences using the Oculus headset. The team employed JavaScript as the primary programming language for its versatility and wide compatibility. The project's structure was facilitated by Vite, a build tool that enhances the development workflow. Additionally, Three.js, a popular JavaScript library, was utilized for managing the 3D aspects of the game, providing a robust foundation for creating interactive and visually appealing virtual environments.

This combination of technologies allowed the team to seamlessly integrate the game with the Oculus headset, providing users with an engaging and immersive experience. The subsequent sections of this report will delve into the specific details of the game's implementation, covering both the technical and creative aspects of our Tower of Hanoi extended reality project.

2 Environment Implementation

To create a captivating Tower of Hanoi extended reality game, the development of the environment involves several crucial elements. At its core, the game requires the creation of disks and a structure to hold them. This basic setup lays the foundation for the Tower of Hanoi puzzle. Building upon this foundation, the team explored ways to enhance the user experience by incorporating a 3D background environment and background music.

2.1 Basic Game Elements: Disks and Structure

The fundamental components of the Tower of Hanoi game are the disks and the structure that supports them. Disks of varying sizes are essential for representing the puzzle's elements, and a solid structure provides the foundation for the puzzle-solving process. In our implementation, we used Three.js, a powerful JavaScript library, to model and render these components in the virtual space. This allowed for seamless interaction and realistic visuals, contributing to an engaging gaming experience.

Now, let's delve into the implementation details of the `Disk` and `MainStructure` classes, showcasing how these classes contribute to the creation and management of the Tower of Hanoi game elements.

2.1.1 Disk Class

To represent the disks within the Tower of Hanoi game, we designed a versatile and reusable `Disk` class using JavaScript and the Three.js library. The `Disk` class encapsulates the properties and behaviors of a disk with a central hole, contributing to the visual and interactive aspects of the game.

Class Structure The `Disk` class is structured to be flexible and easy to use, allowing the creation of disks with various specifications. The constructor function initializes the disk with parameters such as radius, hole radius, height, and the file path for the disk's texture image.

```
/**
 * Class representing a disk with a hole in the center.
 */
export class Disk {
  /**
   * Create a disk with a hole.
   * @param {number} radius - The radius of the disk.
   * @param {number} holeRadius - The hole radius of the disk (default 0.5).
   * @param {number} height - The height of the disk (default 0.5).
   * @param {string} texturePath - The file path for the disk texture image.
   * @param {number} mass - The mass of the disk (default 1).
   */
  constructor(radius, holeRadius = 0.1, height = 0.1, texturePath, mass = 1) {
    this.radius = radius;
    this.holeRadius = holeRadius;
    this.height = height;
    this.texturePath = texturePath;
    this.mass = mass;
    this.createDisk();
    this.setDefaultPosition();
  }

  /**
   * Create the geometry and material for the disk.
   */
  createDisk() {
    const loader = new THREE.TextureLoader();
    const texture = loader.load(this.texturePath);


    // Create the geometry of the disk
    const diskShape = new THREE.Shape();
    diskShape.moveTo(0, 0);
```

```
34      diskShape.absarc(0, 0, this.radius, 0, Math.PI * 2, false);
35
36      // Create the geometry of the hole
37      const holeShape = new THREE.Path();
38      holeShape.moveTo(0, 0);
39      holeShape.absarc(0, 0, this.holeRadius, 0, Math.PI * 2, true);
40
41      // Subtract the hole from the disk geometry
42      diskShape.holes.push(holeShape);
43
44      // Use ExtrudeGeometry to create the extruded geometry
45      const extrudeSettings = {
46        depth: this.height, // Extrusion thickness
47        bevelEnabled: false, // No bevel to get a flat shape
48      };
49
50      const diskGeometry = new THREE.ExtrudeGeometry(diskShape, extrudeSettings);
51
52
53      const diskMaterial = new THREE.MeshBasicMaterial({ map: texture });
54
55
56      // Create the mesh
57      this.mesh = new THREE.Mesh(diskGeometry, diskMaterial);
58      this.mesh.userData.physics = { mass: this.mass };
59
60      // Add edges with a line material
61      const edgesGeometry = new THREE.EdgesGeometry(diskGeometry);
62      const edgesMaterial = new THREE.LineBasicMaterial({ color: 0x000000 });
63      const edges = new THREE.LineSegments(edgesGeometry, edgesMaterial);
64
65      // Add the edges to the main mesh
66      this.mesh.add(edges);
67    }
68
69    setDefaultPosition() {
70      this.mesh.rotation.x = -Math.PI / 2; // -90deg rotation X axis
71      this.mesh.position.set(0, 0, 0);
72    }
73
74    /**
75     * Set the position of the disk.
76     * @param {number} x - The x-coordinate.
77     * @param {number} y - The y-coordinate.
78     * @param {number} z - The z-coordinate.
79     */
80    setPosition(x, y, z) {
81      this.mesh.position.set(x, y, z);
82    }
83
84    /**
85     * Set the rotation of the disk.
86     * @param {number} x - The rotation around the x-axis in radians.
87     * @param {number} y - The rotation around the y-axis in radians.
88     * @param {number} z - The rotation around the z-axis in radians.
89     */
90    setRotation(x, y, z) {
91      this.mesh.rotation.set(x, y, z);
92    }
93
94    /**
95     * Get the height of the disk.
96     * @returns {number} The height of the disk.
97     */
98    getHeightDisk() {
99      return this.height ;
100   }
101 }
```

Listing 1: JS Class for Disks

Geometry and Material Creation The `createDisk` method is responsible for generating the geometry and material for the disk. It uses Three.js to create a disk shape and a hole shape, subtracts the hole from the disk geometry, and then utilizes ExtrudeGeometry to create the final 3D geometry. A texture is applied to the material for a realistic appearance, and edges are added to enhance visualization.

Positioning and Rotation The `setDefaultPosition` method establishes the default rotation and position for the disk. Additionally, the class provides methods (`setPosition` and `setRotation`) for dynamically setting the position and rotation of the disk during runtime.

Height Retrieval To retrieve the height of the disk, the `getHeightDisk` method is available, facilitating interaction with other elements in the Tower of Hanoi game.

This modular approach allows for the easy integration of disks into the game, providing a foundation for creating an immersive and visually appealing Tower of Hanoi extended reality experience.

```
1 // Example Usage
2 const disk1 = new Disk(1, 0.2, 0.1, 'path/to/texture.jpg');
3 disk1.setPosition(0, 0, 0);
```

Listing 2: JS Example for Disk

In this example, a disk is instantiated with specific parameters, and its position is set to (0, 0, 0). The `Disk` class provides a clean and efficient way to manage the visual aspects of the Tower of Hanoi game.

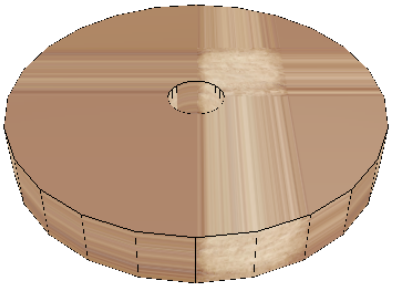

Figure 1: Instance of Disk

2.1.2 MainStructure Class

The `MainStructure` class is responsible for creating the central structure in the Tower of Hanoi game, consisting of a base rectangle and three cylinders. Let's focus on the `createMainStructure` method, which initializes and positions these components.

Class Structure The `createMainStructure` method is a crucial part of the `MainStructure` class, where the geometry and material for the main structure are generated. Here's a detailed breakdown of the method:

```
1
2 export class MainStructure {
3   /**
4    * Create the main structure with a base rectangle and three cylinders on top.
5    * @param {number} baseWidth - The width of the base rectangle.
```

```
   * @param {number} baseDepth - The depth of the base rectangle.
   * @param {number} baseHeight - The height of the base rectangle.
   * @param {number} cylinderRadius - The radius of the cylinders.
   * @param {number} cylinderHeight - The height of the cylinders.
   * @param {string} baseTexturePath - The file path for the base texture image.
   * @param {string} cylinderTexturePath - The file path for the cylinder texture image.
   */
  constructor(baseWidth, baseDepth, baseHeight, cylinderRadius, cylinderHeight, baseTexturePath, cylinderTexturePath) {
    this.baseWidth = baseWidth;
    this.baseDepth = baseDepth;
    this.baseHeight = baseHeight;
    this.cylinderRadius = cylinderRadius;
    this.cylinderHeight = cylinderHeight;
    this.baseTexturePath = baseTexturePath;
    this.cylinderTexturePath = cylinderTexturePath;

    this.createMainStructure();
    this.setDefaultPosition();
  }

  /**
  * Create the geometry and material for the main structure.
  */
  createMainStructure() {
    const loader = new THREE.TextureLoader();

    // Calculate the extended width of the base rectangle
    const extendedWidth = this.baseWidth + 2 * this.cylinderRadius;

    // Calculate the width of each section
    const sectionWidth = this.baseWidth / 3;

    // Load textures
    const baseTexture = loader.load(this.baseTexturePath);
    const cylinderTexture = loader.load(this.cylinderTexturePath);

    // Create the base rectangle geometry with extended sides
    const baseGeometry = new THREE.BoxGeometry(extendedWidth, this.baseHeight, this.baseDepth);
    const baseMaterial = new THREE.MeshBasicMaterial({ map: baseTexture });
    this.baseMesh = new THREE.Mesh(baseGeometry, baseMaterial);
    this.baseMesh.userData.physics = { mass: 1 }; // Set mass for the base

    // Create the cylinder geometry
    const cylinderGeometry = new THREE.CylinderGeometry(this.cylinderRadius, this.cylinderRadius, this.cylinderHeight, 32);
    const cylinderMaterial = new THREE.MeshBasicMaterial({ map: cylinderTexture });

    // Create cylinders and position them in the middle of each section
    this.cylinder1 = new THREE.Mesh(cylinderGeometry, cylinderMaterial);
    this.cylinder1.position.set(-extendedWidth / 2 + this.cylinderRadius + sectionWidth / 2, this.baseHeight / 2 + this.cylinderHeight / 2, 0);
    this.cylinder1.userData.physics = { mass: 1 }; // Set mass for the cylinder

    this.cylinder2 = new THREE.Mesh(cylinderGeometry, cylinderMaterial);
    this.cylinder2.position.set(-extendedWidth / 2 + this.cylinderRadius + sectionWidth + sectionWidth / 2, this.baseHeight / 2 + this.cylinderHeight / 2, 0);
    this.cylinder2.userData.physics = { mass: 1 }; // Set mass for the cylinder

    this.cylinder3 = new THREE.Mesh(cylinderGeometry, cylinderMaterial);
    this.cylinder3.position.set(-extendedWidth / 2 + this.cylinderRadius + 2 * sectionWidth + sectionWidth / 2, this.baseHeight / 2 + this.cylinderHeight / 2, 0);
    this.cylinder3.userData.physics = { mass: 1 }; // Set mass for the cylinder

    // Create a group to hold all the objects
    this.mainStructure = new THREE.Group();
```

```
67      this.mainStructure.add(this.baseMesh);
68      this.mainStructure.add(this.cylinder1);
69      this.mainStructure.add(this.cylinder2);
70      this.mainStructure.add(this.cylinder3);
71    }
72  // Getter and Setter ...
```

Listing 3: JS Class for MainStructure

- **Texture Loading:** The method starts by loading textures for both the base rectangle and the cylinders using the Three.js `TextureLoader`.
- **Base Rectangle Creation:** It creates a base rectangle geometry (`baseGeometry`) with extended sides to accommodate the cylinders. The material for the base (`baseMaterial`) is mapped with the loaded base texture.
- **Cylinder Creation:** Three cylinders (`cylinder1`, `cylinder2`, and `cylinder3`) are created with appropriate positions. The cylinders are positioned in the middle of each section of the extended base rectangle.
- **Physics Properties:** Physics properties (mass) are set for both the base and cylinders, which is useful for physics simulations.
- **Grouping Objects:** All created objects, including the base rectangle and cylinders, are grouped together using a Three.js `Group` (`mainStructure`).

This method ensures that the main structure is created with the correct geometry, material, and positioning, laying the foundation for an integral part of the Tower of Hanoi extended reality game.

```
1  // Example Usage
2  const mainStructure = new MainStructure(5, 1, 0.1, 0.05, 0.7, 'path/to/texture_base
       .jpg', 'path/to/texture_cylinder.jpg')
3  mainStructure.setPosition(0, 0, 0);
```

Listing 4: JS Example for MainStructure

Figure 2: Instance of MainStructure

2.1.3 Board Class

To manage the overall presentation and interaction in the Tower of Hanoi game, we implemented the `Board` class using JavaScript and the Three.js library. The `Board` class encompasses the visual representation of the game board, including buttons for essential actions and dynamic text to display moves and status.

Class Structure The `Board` class is designed to provide a customizable game board with default parameters for color, height, and width. It includes buttons for actions such as exit, reset, and play/pause, each with associated text for user guidance. The class maintains a count of moves and status information to enhance the gaming experience.

```
1  export class Board {
2    constructor(color = 'grey', boardHeight = 1, boardWidth = 1) {
3        this.color = color;
4        this.boardHeight = boardHeight;
```

```
5       this.boardWidth = boardWidth;
6       this.moves = 0; // Initialize moves
7       this.status = 'valid'; // Initialize status
8
9       // Create the board geometry and material
10      const boardGeometry = new THREE.PlaneGeometry(this.boardWidth, this.
            boardHeight);
11      const boardMaterial = new THREE.MeshBasicMaterial({ color: this.color, side:
            THREE.DoubleSide });
12      this.board = new THREE.Mesh(boardGeometry, boardMaterial);
13
14      // Create buttons
15      this.createButtons();
16      this.createText();
17  }
18
19  createButtons() {
20      // Define button dimensions
21      const buttonWidth = 0.6;
22      const buttonHeight = 0.2;
23      const buttonDepth = 0.1;
24      const buttonGeometry = new THREE.BoxGeometry(buttonWidth, buttonHeight,
            buttonDepth);
25
26      // Example: Exit Button
27      const exitMaterial = new THREE.MeshBasicMaterial({ color: 0xff0000 }); // red
             color
28      this.exit = new THREE.Mesh(buttonGeometry, exitMaterial);
29      this.exit.position.set(-this.boardWidth / 2 + buttonWidth, this.boardHeight /
             2 - buttonHeight, 0.1);
30
31
32      // Example: Reset Button
33      const resetMaterial = new THREE.MeshBasicMaterial({ color: 0x0000ff }); //
            blue color
34      this.reset = new THREE.Mesh(buttonGeometry, resetMaterial);
35      this.reset.position.set(-this.boardWidth / 2 + 3 * buttonWidth, this.
            boardHeight / 2 - buttonHeight, 0.1);
36
37
38      const playMaterial = new THREE.MeshBasicMaterial({ color: 0x0000ff }); //
            blue color
39      this.play = new THREE.Mesh(buttonGeometry, playMaterial);
40      this.play.position.set(-this.boardWidth / 2 + 5 * buttonWidth, this.
            boardHeight / 2 - buttonHeight, 0.1);
41
42      const loader = new FontLoader();
43      loader.load('./assets/helvetiker_regular.typeface.json', (font)=>{
44          const textOptions = {
45              font: font,
46              size: 0.05,
47              height: 0.05,
48          };
49      // Create text geometry for each button
50      const exitTextGeometry = new TextGeometry('Exit', textOptions);
51      const resetTextGeometry = new TextGeometry('Reset', textOptions);
52      const play_pauseTextGeometry = new TextGeometry('Play/Pause', textOptions);
53
54      // Create meshes for the text
55      const textMaterial = new THREE.MeshBasicMaterial({ color: 0xffffff });
56      const exitText = new THREE.Mesh(exitTextGeometry, textMaterial);
57      const resetText = new THREE.Mesh(resetTextGeometry, textMaterial);
58      const playPauseText = new THREE.Mesh(play_pauseTextGeometry, textMaterial);
59
60      // Position the text on the buttons
61      exitText.position.set(-0.1, -0.05, 0.05); // Adjust these values as needed
62      resetText.position.set(-0.1, -0.05, 0.05);
63      playPauseText.position.set(-0.1, -0.05, 0.05);
64
65      // Add the text to the buttons
```

```
66        console.log(this.exit)
67        this.exit.add(exitText);
68        this.reset.add(resetText);
69        this.play.add(playPauseText);
70        });
71
72        this.board.add(this.exit);
73        this.board.add(this.reset);
74        this.board.add(this.play);
75    }
76
77    createText() {
78          const loader = new FontLoader();
79          loader.load('./assets/helvetiker_regular.typeface.json', (font) => {
80            this.font = font; // Store the font for later use
81            this.updateMovesAndStatus(0, 'valid'); // Initial text setup
82        });
83      }
84
85      updateMovesAndStatus(moves, invalid) {
86        this.moves = moves;
87          if (invalid == true){
88            this.status = 'invalid';
89          }
90          else{
91            this.status = 'valid';
92          }
93
94        const textOptions = {
95            font: this.font,
96            size: 0.3,
97            height: 0.05,
98        };
99
100       // Update moves text
101       if (this.movesText) this.board.remove(this.movesText); // Remove old text
102       const movesTextGeometry = new TextGeometry('Moves: ' + this.moves,
              textOptions);
103       const movesTextMaterial = new THREE.MeshBasicMaterial({ color: 0xffffff });
104       this.movesText = new THREE.Mesh(movesTextGeometry, movesTextMaterial);
105       this.movesText.position.set(-this.boardHeight / 2, 0, 0);
106       this.board.add(this.movesText);
107
108       // Update status text
109       if (this.statusText) this.board.remove(this.statusText); // Remove old text
110       const statusTextGeometry = new TextGeometry('Status: ' + this.status,
              textOptions);
111       const statusTextMaterial = new THREE.MeshBasicMaterial({ color: 0xffffff });
112       this.statusText = new THREE.Mesh(statusTextGeometry, statusTextMaterial);
113       this.statusText.position.set(-this.boardHeight / 2, -this.boardWidth / 4, 0);
114       this.board.add(this.statusText);
115   }
116 }
```

Listing 5: JS Class for Board

Button and Text Creation The `createButtons` method generates interactive buttons on the board, such as exit, reset, and play/pause. These buttons include 3D geometries with associated materials for visual appeal. Text labels are added to the buttons using the Three.js library's capabilities.

Text Display and Updates The `createText` method initializes the font for later use in displaying text on the board. The `updateMovesAndStatus` method dynamically updates and displays the move count and game status. This interactive text provides valuable feedback to the player during gameplay.

```
1 // Example Usage
```

```
const gameBoard = new Board('blue', 2, 2);
gameBoard.updateMovesAndStatus(5, false);
```

Listing 6: JS Example for Board

In this example, a game board is instantiated with specific parameters, and the moves and status are updated during gameplay. The `Board` class enhances the Tower of Hanoi game interface, combining visual elements and interactivity for an engaging user experience.

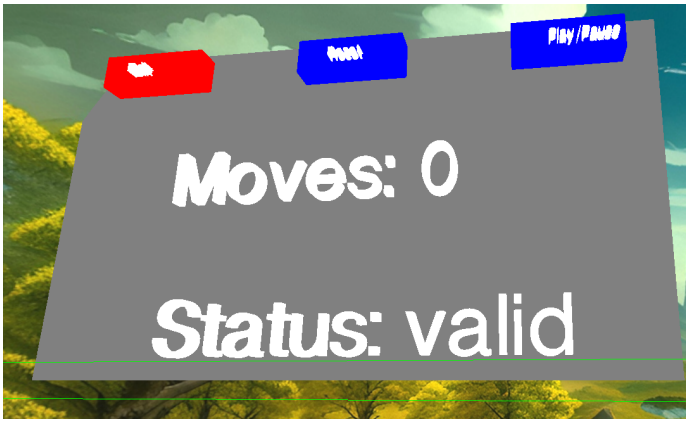

Figure 3: Instance of Board

2.2 3D Background Environment

To enhance the immersive quality of the Tower of Hanoi game, we implemented a 3D background environment using the GLTFLoader in Three.js. This addition introduces an extra layer to the scene, providing a sense of depth and presence for a more engaging visual experience.

```
setUpBackground() {
    const loader = new GLTFLoader();
    loader.load('../../assets/models/back.glb',
        (gltf) => {
            const model = gltf.scene;
            model.position.z = 1;
            this.scene.add(model);
        },
        (xhr) => {
            // Progress callback (optional)
            // console.log((xhr.loaded / xhr.total * 100) + '% loaded');
        },
        (error) => {
            console.error('Error loading the model:', error);
        }
    );
}
```

Listing 7: JS Code for Including 3D Background

Code Explanation:

- **Loader Initialization:** An instance of the GLTFLoader is created, which is a loader specifically designed for loading models in the glTF format, a popular format for 3D models.
- **Model Loading:** The loader loads the 3D model from the specified file path ('../../assets/models/back.glb'). Upon successful loading, the provided callback function is executed, receiving a glTF object (`gltf`).

- **Scene Placement:** The 3D model is retrieved from the glTF object (`gltf.scene`). The model's position is set to have a positive z-coordinate (`model.position.z = 1`), placing it slightly in front of the camera in the scene.
- **Adding to Scene:** The model is added to the main scene (`this.scene`), integrating it as part of the overall 3D environment.
- **Progress Callback (Optional):** During the loading process, a progress callback is provided (commented out in this case). It can be used to track the loading progress if needed.
- **Error Handling:** If there is an error during the loading process, an error message is logged to the console (`console.error('Error loading the model:', error)`).

This code segment ensures that the 3D background model is loaded and positioned appropriately within the Three.js scene, contributing to the immersive quality of the Tower of Hanoi game.

Here is a part of the background:

Figure 4: Part of the 3D Background

2.3 Background Music

To enrich the gaming experience, we introduced a carefully curated background soundtrack to the Tower of Hanoi extended reality game. The background music enhances the immersive atmosphere, providing players with a multi-sensory experience that complements the gameplay.

```
setUpAudio() {
    // Create an audio listener attached to the camera
    const listener = new THREE.AudioListener();
    this.camera.add(listener);

    // Create a global audio object
    const audio = new THREE.Audio(listener);

    // Create an AudioLoader
    const audioLoader = new THREE.AudioLoader();

    // Load the soundtrack
    audioLoader.load('../../assets/sakura.mp3', (buffer) => {
        // Set the audio buffer and configure playback properties
        audio.setBuffer(buffer);
        audio.setLoop(true); // Loop the soundtrack
        audio.setVolume(2); // Adjust the volume (2 times the normal volume)
        audio.play(); // Start playing the soundtrack
    });
}
```

Listing 8: JS code for adding Audio

Code Explanation:

- **Audio Listener Creation:** An audio listener is created using `THREE.AudioListener()`, and it is attached to the camera to ensure that the audio is spatialized based on the camera's position.
- **Global Audio Object:** A global audio object (`audio`) is created using the audio listener.
- **AudioLoader Initialization:** An `AudioLoader` is instantiated to load the soundtrack asynchronously.
- **Soundtrack Loading:** The `audioLoader.load()` method loads the soundtrack from the specified file path ('../../assets/sakura.mp3'). Upon successful loading, the provided callback function is executed, receiving an `AudioBuffer` (`buffer`).
- **Audio Configuration:** The audio buffer is set for the `audio` object, and playback properties are configured. The soundtrack is set to loop indefinitely, and its volume is adjusted (2 times the normal volume).
- **Start Playback:** The `audio.play()` method is called to start playing the background soundtrack.

This code segment ensures the seamless integration of background music into the Tower of Hanoi extended reality game, contributing to a more immersive and enjoyable gaming experience.

2.4 Final Environment

Figure 5: Final Environment with 3 Disks, Main Structure, Background, and Music.

3 Selection and manipulation techniques

3.1 Raycaster for Object Selection

The **Raycaster** serves as a fundamental tool in enabling precise object selection within the virtual environment. In the Tower of Hanoi VR game, it plays a key role in detecting user interactions with game elements. By casting a virtual ray into the scene, the Raycaster identifies the object or objects intersected by the ray, allowing for targeted selection.

```
    //Raycaster Initialization
    raycaster = new THREE.Raycaster();
    raycaster.params.Line.threshold = 3;
    ...
    //Use of raycaster to allow player to drag the disk
    scene.controls.addEventListener('dragstart', function (event) {
    // Disable orbit control during drag
    makingMove = true;
    orbitControls.enabled = false;
    selectedObject = event.object;

    // Check for intersecting disks during drag start
    const intersects_disks = raycaster.intersectObjects( scene.disks_mashes, true )
        ;
    if ( intersects_disks.length > 0 ) {
        intersects_disks[0].object.material.color.set( 0xffb3b3 ); // Change color
            of selected object
        let disk_onTop = hasDisksOnTop(intersects_disks[0].object,scene.
            disks_mashes);
    ...}
```

Listing 9: Use of Raycaster

To enhance **player feedback** and facilitate easier interaction with the environment, the color of the disks intersected by the raycaster is altered.

3.2 Drag and Drop Implementation with DragControls

To enhance user interaction, we implemented the drag and drop functionality using **DragControls** in Three.js. This feature allows users to select and move game objects seamlessly within the 3D environment.

```
export class DraggablesManager {
  constructor(objects, camera, renderer, scene) {
    this.objects = objects;
    this.dragControls = new DragControls(this.objects, camera, renderer.domElement)
  }
  addObject(object) {
    this.objects.push(object);
    this.dragControls.setObjects(this.objects);
  }
  onHoverOn(event) {
    // Add your logic when hovering on an object
    const hoveredObject = event.object;
    console.log('Hovered on:', hoveredObject);
  }
  onHoverOff(event) {
    // Add your logic when hovering off an object
    const unhoveredObject = event.object;
    console.log('Hovered off:', unhoveredObject);
  }
}
```

Listing 10: JS Class for Drag Control

Code Explanation:

- **DragControls Initialization:** We initialized the DragControls to enable the drag and drop functionality. This included creating an instance of DragControls, associating it with the camera and the array of draggable objects.
- **Event Handling:** We set up event listeners to handle the start, drag, and end events triggered by DragControls. These events enabled us to update the position of the dragged object during interaction.
- **Selectable Objects:** We defined the array of selectable objects to include the disks in the Tower of Hanoi game. These objects became draggable and responsive to user input.

This implementation provides an intuitive way for users to interact with the Tower of Hanoi game by selecting and moving objects using drag and drop functionality.

4 Physics Implementation

The implementation of physics plays a crucial role in creating a realistic and interactive user experience. Leveraging the **enable3d** library, which integrates the Ammo.js physics engine with Three.js, we were able to introduce dynamic, static, and kinematic bodies, along with various features such as compound shapes, constraints, CCD motion clamping, and more. This chapter delves into the key aspects of the physics implementation, illustrating how it enhances the overall realism and interactivity of the game.

4.1 Enabling Physics with "Enable3d" and "Ammo.js"

The foundation of the physics implementation lies in the integration of Ammo.js through enable3d. **Ammo.js** is a direct JavaScript port of the popular C++ physics engine called Bullet Physics. Bullet Physics is widely used in the gaming industry and is known for its stability, performance, and extensive set of features. he AmmoPhysics class facilitates the addition of physics to the Three.js scene, providing a robust framework for simulating real-world interactions. With enable3d, we effortlessly introduced physics to static elements, such as the game table, as well as dynamic elements, like the Tower of Hanoi disks.

```
...
import * as ENABLE3D from '@enable3d/ammo-physics';

...
const clock = new THREE.Clock()
let physics;
const { AmmoPhysics, PhysicsLoader } = ENABLE3D
...
//Threescene is the main script
const ThreeScene = () => {
    //INIT
    scene = new Scene();
    physics = new AmmoPhysics(scene.scene);
    CURRENTCOLOR = scene.disks_mashes[0].material.color.getHex();
    physics = new AmmoPhysics(scene.scene);
    //Add physics to the scene
    physics.debug.enable();
    ...}

....

//Rendering of physics
function render(event) {
    physics.update(clock.getDelta() * 1000);
    physics.updateDebugger();
    scene.renderer.render(scene.scene, scene.camera);
    // ... (existing logic for updating moves and status)
}

//...


//The PhysicsLoader takes care of loading the wasm and js files of Ammo.js before
    starting the script.
window.addEventListener('DOMContentLoaded', () => {
    PhysicsLoader('/ammo', () => ThreeScene())
  })
```

Listing 11: Ammophyiscs and PhyisicsLoader

The *physics debugger* helps us see and understand the shapes controlled by the physics engine, making it easier to understand the collisions between the various objects in the scene.

4.2 Static Ground and Disks implementation

To establish a stable foundation for the game, a static ground was introduced.
This ground, representing the game table, was created with zero mass to ensure it remains stationary throughout the game.

```
physics.add.existing(scene.table, { name: "table", mass: 0 });
```

Dynamic bodies were applied to each disk, allowing them to respond to external forces such as gravity and user interactions. The offset property was utilized to fine-tune the position of the disks within the physics simulation.

```
physics.add.existing(scene.disk1.mesh, { mass: 1, offset: { y: -0.05 } });
physics.add.existing(scene.disk2.mesh, { mass: 1, offset: { y: -0.05 } });
physics.add.existing(scene.disk3.mesh, { mass: 1, offset: { y: -0.05 } });
```

The main advantage of the enable3d library is the presence of numerous predefined shapes that can be used to represent objects in the physical world. Some of these shapes are:

- Box Shape: represents a rectangular prism or cube.
- Sphere Shape: represents a spherical object.
- Cylinder Shape: represents a cylindrical object.
- Cone Shape: represents a cone-shaped object.
- Capsule Shape: combines a cylinder and two hemispheres, resembling a capsule.
- Convex Hull Shape: represents a convex hull, providing a simplified representation of a complex mesh.
- Compound Shape: allows combining multiple shapes to create more complex geometries.
- Trimesh Shape: represents a triangle mesh.

Compound shapes in enable3d provide the option to utilize automatically generated shapes by specifying the "shape" parameter as "hacd," "convexMesh," or "concaveMesh."

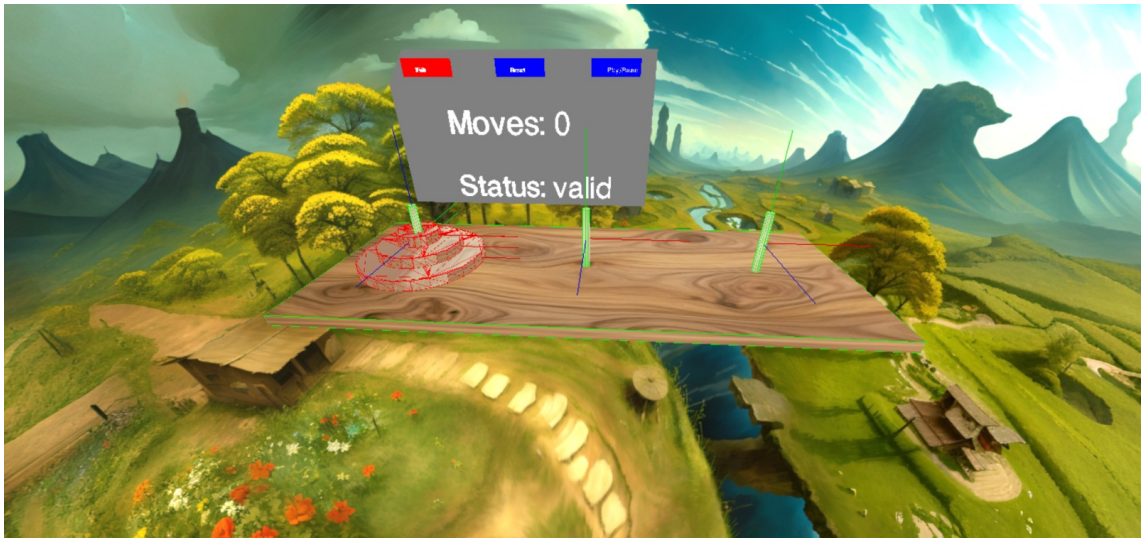

Figure 6: View of physics debug tool

4.3 Physics Constraints

The control of the **setCollisionFlag** is a pivotal aspect of managing the physics interactions. This flag determines whether an object within the physics simulation is allowed to collide with other objects or remain static.
When a user initiates a move, such as dragging a Tower of Hanoi disk, the setCollision flag is manipulated to control its behavior. Initially set to 2 during the dragstart event, this flag designates the disk as movable, allowing it to interact dynamically with the environment. Once the user completes the move, during the dragend event, the flag is reset to 0, indicating that the disk should no longer respond to external forces, effectively making it static.
Additionally, we leverage the setCollision flag when interacting with the game table, which serves as the static ground. If a disk collides with the table, the collision event is detected, and the disk's setCollision flag is set to 1, rendering it static and preventing further movement.
This controlled manipulation of the setCollision flag ensures a seamless and intuitive user experience, allowing for dynamic interactions during user-initiated moves while maintaining stability and preventing unintended collisions with static elements in the virtual environment.

5 Rule Implementation

In the Tower of Hanoi game, the implementation of rules is crucial for ensuring a challenging and engaging experience. One of the key functions responsible for enforcing game rules is the `checkCollisions` function.

5.1 Collision Detection

The `checkCollisions` function is responsible for detecting collisions between disks during gameplay. It utilizes the bounding box technique to determine if any two disks intersect. The bounding box for each disk is updated based on its current position and orientation in the 3D space.

```
function checkCollisions() {
    scene.disk1BB.copy(scene.disks[0].mesh.geometry.boundingBox).applyMatrix4(scene
        .disks[0].mesh.matrixWorld);
    scene.disk2BB.copy(scene.disks[1].mesh.geometry.boundingBox).applyMatrix4(scene
        .disks[1].mesh.matrixWorld);
    scene.disk3BB.copy(scene.disks[2].mesh.geometry.boundingBox).applyMatrix4(scene
        .disks[2].mesh.matrixWorld);

    // Check for collisions between Disk 1 and Disk 2
    if (scene.disk1BB.intersectsBox(scene.disk2BB)) {
        if (scene.disks[0].mesh.position.y > scene.disks[1].mesh.position.y) {
            Invalid = true;
        }
    }

    // Check for collisions between Disk 1 and Disk 3
    if (scene.disk1BB.intersectsBox(scene.disk3BB)) {
        if (scene.disks[0].mesh.position.y > scene.disks[2].mesh.position.y) {
            Invalid = true;
        }
    }
    // Check for collisions between Disk 2 and Disk 3
    if (scene.disk2BB.intersectsBox(scene.disk3BB)) {
        if (scene.disks[1].mesh.position.y > scene.disks[2].mesh.position.y) {
            Invalid = true;
        }
    }
}
```

Listing 12: Collision Detection Function

In the code snippet above, the function checks for collisions between each pair of disks (Disk 1 and Disk 2, Disk 1 and Disk 3, Disk 2 and Disk 3) and updates the game state accordingly.

This collision detection mechanism adds a layer of complexity to the Tower of Hanoi game, requiring players to adhere to the rules and make strategic moves to successfully complete the puzzle.

5.2 Drag-and-Drop Interaction

Enabling a seamless drag-and-drop interaction for the Tower of Hanoi game enhances the user experience and allows players to make strategic moves. The implementation involves utilizing event listeners and controls provided by the Three.js library.

```
    scene.controls.addEventListener( 'drag', function(event){
        if (selected?.body.getCollisionFlags() === 2) {
            const { x, y } = pointer
            const speed = 0.03
            const movementX = (x - prev.x) * speed
            const movementZ = (y - prev.y) * -speed
            selected.position.x += movementX
            selected.position.z += movementZ
            selected.body.needUpdate = true
            prev = { x, y }
        }
        selectedObject = event.object
        render();
    } );

document.addEventListener( 'pointermove', onPointerMove );
document.addEventListener( 'pointerdown', onPointerDown );

scene.controls.addEventListener('dragstart', function (event) {
    // Disable orbit control during drag
    makingMove = true;
    orbitControls.enabled = false;
    selectedObject = event.object;

    // Check for intersecting disks during drag start
    const intersects_disks = raycaster.intersectObjects( scene.disks_mashes, true )
        ;
    if ( intersects_disks.length > 0 ) {
        intersects_disks[0].object.material.color.set( 0xffb3b3 ); // Change color
            of selected object
        let disk_onTop = hasDisksOnTop(intersects_disks[0].object,scene.
            disks_mashes);
        console.log("disk on top variable: ",disk_onTop);

        if(disk_onTop === false){
            selected = intersects_disks[0].object;
            console.log("selected: ",selected.body.getCollisionFlags());
            selected.body.setCollisionFlags(2); // Set collision flag for the
                dragged disk
        }
    }
});

scene.controls.addEventListener('dragend', function (event) {
    // Enable orbit control after drag ends
    makingMove = false;
    orbitControls.enabled = true;
    const selectedObject = event.object;

    // Check if the dragged object is a disk
    if (selectedObject.body.getCollisionFlags() === 2){
        movesMade +=1; // Increment move count
    }

    selectedObject.material.color.setHex( CURRENTCOLOR );
    selectedObject.body.setCollisionFlags(0); // Reset collision flag

    // When the object is dropped, check if it is on the table
    // If on the table, set the collision flag to 1 (static)
```

```
56      selectedObject.body.on.collision((otherObject, event) => {
57          if (otherObject.name === 'table') selectedObject.body.setCollisionFlags(1);
58      })
59
60      selected = null; // Reset selected object
61  });
```

Listing 13: Drag-and-Drop Interaction

In the provided code snippet, the drag-and-drop interaction is handled through event listeners and controls. The drag, dragstart, and dragend events are utilized to manage the state of the interaction, enable/disable orbit control, and update relevant parameters.

The drag-and-drop feature enhances player control and adds an interactive layer to the Tower of Hanoi game.

5.3 Rendering and Game State Update

The rendering process in the Tower of Hanoi game plays a crucial role in updating the game state, including the move count and game status. The `render` function is responsible for orchestrating these updates, ensuring a smooth and responsive gaming experience.

```
1   function render(event) {
2       // Check for collisions if not in the process of making a move
3       if (makingMove == false) {
4           checkCollisions();
5       }
6
7       // Update physics simulation
8       physics.update(clock.getDelta() * 1000);
9       physics.updateDebugger();
10
11      // Render the scene
12      scene.renderer.render(scene.scene, scene.camera);
13
14      // Update the game board with the latest move count and status
15      scene.gameBoard.updateMovesAndStatus(movesMade, Invalid);
16  }
17
18  // Initial update of the game board state
19  scene.gameBoard.updateMovesAndStatus(movesMade, Invalid);
```

Listing 14: Rendering and Game State Update

In the code snippet above, the `render` function is described as the central hub for updating the game state during the rendering process. It checks for collisions, updates the physics simulation, renders the scene, and ensures that the game board reflects the latest move count and status.

This integration of rendering and game state update contributes to a responsive and visually informative Tower of Hanoi gaming experience.

5.4 Scenegraph :

The provided scene graph offers a streamlined view of the Tower of Hanoi game's architecture. Central to the graph is the 'Scene' node, which anchors all components of the game world. The 'Board' and 'MainStructure' nodes represent the core gameplay elements, while the 'Disk' nodes are the movable objects that players will interact with.

Interactive elements, such as the 'Play', 'Reset', and 'Moves' buttons, are directly tied to the game logic, allowing for user input to dictate game flow. Accompanying text nodes provide essential feedback and instructions to the player.

The lighting nodes, 'Ambient Light' and 'Top Light', define the game's visual atmosphere, influencing the overall aesthetic and playability. The 'Background' is loaded via a 'GLTFLoader', indicating the use of advanced graphical assets to enrich the visual setting.

This hierarchical setup is integral to the game's functionality, dictating the rendering order and the relationship between different game elements, which is vital for both the game's performance and the user experience.

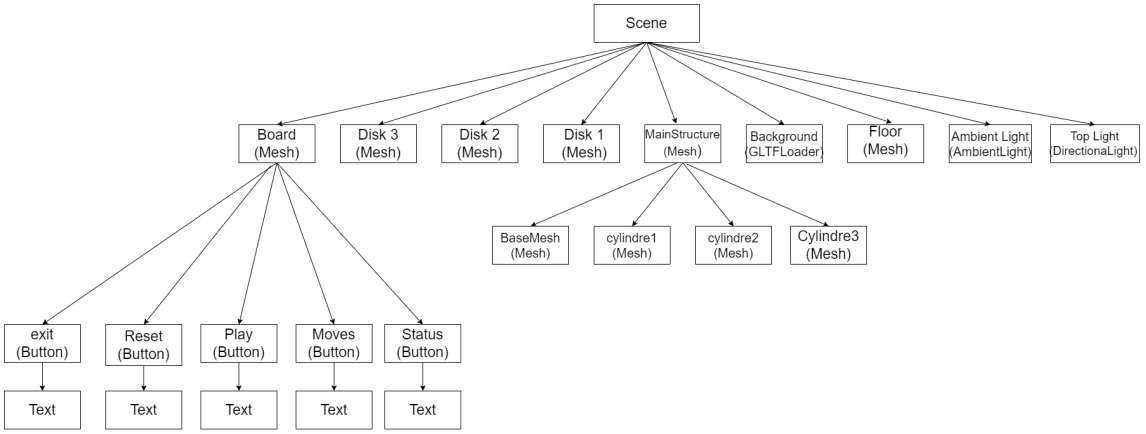

Figure 7: scengraphe