# 3D Shape Game Project Report

# Index

1. Project Overview

2. Backend Implementation

3. Frontend Implementation

4. Development Thought Process

5. Results and Screenshots

## 1. Project Overview

The 3D Shape Game is an ambitious multiplayer web application designed to allow users to create, manipulate, and interact with 3D shapes in a shared virtual space. The project aims to blend creativity with real-time collaboration, pushing the boundaries of web-based 3D interactions.

Project Objectives:

* Implement intuitive 2D shape drawing functionality
* Enable seamless 3D extrusion of 2D shapes
* Create a responsive shared multiplayer environment
* Facilitate real-time interaction between players

Technologies Used:

* Backend: Node.js, TypeScript, Colyseus, Express
* Frontend: Angular, TypeScript, BabylonJS, Colyseus.js

Current Status: As of now, I have successfully implemented the 2D shape drawing and 3D extrusion features. These core functionalities provide a solid foundation for the game. However, the movement of 3D objects and real-time synchronization of other players' shapes are still in progress.

## 2. Backend Implementation

Design Choices:

1. Colyseus Framework: I opted for Colyseus as it was mentioned in the task and due to its robust real-time multiplayer capabilities and seamless integration with TypeScript. Its room-based architecture aligns perfectly with our game design.
2. Room-based Architecture: The game utilizes a room-based system, allowing multiple game instances to run simultaneously. This design choice enhances scalability and provides a structure for future implementation of different game modes.
3. State Synchronization: I implemented automatic state synchronization to ensure all clients have up-to-date game information. This approach reduces the complexity of manual data management and minimizes the risk of state inconsistencies across clients.

Implementation and Code Analysis:

Key Components:

* + GameRoom: Manages player connections, actions, and state updates.
  + GameState: Defines the structure of the shared game state.

Code Snippet (GameRoom.ts):

export class GameRoom extends Room<GameState> {

onCreate(options: any) {

this.setState(new GameState());

this.onMessage("draw", (client, message) => {

const player = this.state.players.get(client.sessionId);

if (player) {

player.shape = message.points;

console.log(`Player ${client.sessionId} drew a shape`);

}

});

this.onMessage("extrude", (client, message) => {

const player = this.state.players.get(client.sessionId);

if (player) {

player.height = message.height;

console.log(`Player ${client.sessionId} extruded shape to height ${message.height}`);

}

});

}

onJoin(client: Client, options: any) {

this.state.players.set(client.sessionId, new Player());

console.log(`Player ${client.sessionId} joined`);

}

onLeave(client: Client, consented: boolean) {

this.state.players.delete(client.sessionId);

console.log(`Player ${client.sessionId} left`);

}

}

This structure allows for easy handling of different game actions and player management. The use of TypeScript enhances code readability and helps catch potential errors early in the development process.

## 3. Frontend Implementation

Design Choices:

1. Angular Framework: I chose Angular for its powerful component-based architecture and TypeScript support. Its robust ecosystem and tooling support significantly accelerated our development process.
2. BabylonJS: Selected for its robust 3D rendering capabilities and excellent performance in web browsers. Its extensive documentation and active community were key factors in this decision.
3. Component Structure: I had divided the UI into distinct components (Game, Drawing, Controls) for better organization, reusability, and maintenance. This structure allows for easier testing and future enhancements.

Implementation and Code Analysis:

Key Components:

1. GameComponent: Manages the 3D scene and overall game state.
2. DrawingComponent: Handles 2D shape drawing.
3. ControlsComponent: Provides UI for shape manipulation.

Code Snippet (game.component.ts):

export class GameComponent implements OnInit, AfterViewInit {

@ViewChild('renderCanvas') private canvasRef: ElementRef;

private scene: BABYLON.Scene;

private engine: BABYLON.Engine;

constructor(private gameService: GameService) {}

ngOnInit() {

this.gameService.joinGame().then(() => {

console.log('Connected to game server');

}).catch(error => {

console.error('Failed to connect to game server', error);

});}

ngAfterViewInit() {

this.createScene();

this.engine.runRenderLoop(() => {

this.scene.render();

});

}

private createScene() {

this.engine = new BABYLON.Engine(this.canvasRef.nativeElement, true);

this.scene = new BABYLON.Scene(this.engine);

const camera = new BABYLON.ArcRotateCamera("camera", -Math.PI / 2, Math.PI / 2.5, 10, BABYLON.Vector3.Zero(), this.scene);

camera.attachControl(this.canvasRef.nativeElement, true);

const light = new BABYLON.HemisphericLight("light", new BABYLON.Vector3(0, 1, 0), this.scene);

// Add ground and basic environment setup

}

updateShape(shapeData: any) {

// Logic to update or create 3D shape based on 2D drawing

const shape = BABYLON.MeshBuilder.ExtrudeShape("shape", {

shape: shapeData.points.map(p => new BABYLON.Vector3(p.x, p.y, 0)),

path: [new BABYLON.Vector3(0, 0, 0), new BABYLON.Vector3(0, shapeData.height, 0)],

}, this.scene);

}}

This structure allows for clear separation of concerns between 3D rendering, 2D drawing, and game logic, enhancing maintainability and allowing for easier future expansions.

## 4. Development Thought Process

1. Backend Thought Process:

1. State Synchronization Optimization: Initially, I naively synchronized the entire game state with every change, leading to performance issues as the number of players and objects increased. I tackled this by implementing a delta-based synchronization system, only sending changes rather than the entire state. This significantly reduced network traffic and improved responsiveness.  
     
   Scenarios  
     
   // Before

this.state.players.forEach((player) => {

// Send entire player state

});

// After

this.state.players.forEach((player) => {

if (player.changed()) {

// Send only changed properties

}

});

1. Scalability and Room Management - As I had added more features, the GameRoom class became unwieldy. I refactored it to use a command pattern, separating different game actions into their own handler classes. This not only improved code organization but also made it easier to add new features and debug existing ones.
2. Concurrency and Race Conditions – I had encountered issues with concurrent modifications to the game state, particularly when multiple players performed actions simultaneously. To address this, I implemented a queue-based action processing system, ensuring that actions are processed in the order they are received, maintaining game state consistency.
3. Error Handling and Logging - Initially, error handling was minimal, making it difficult to diagnose issues in production. I implemented a comprehensive error handling and logging system, using a combination of try-catch blocks and a custom logger that could be easily configured for different environments

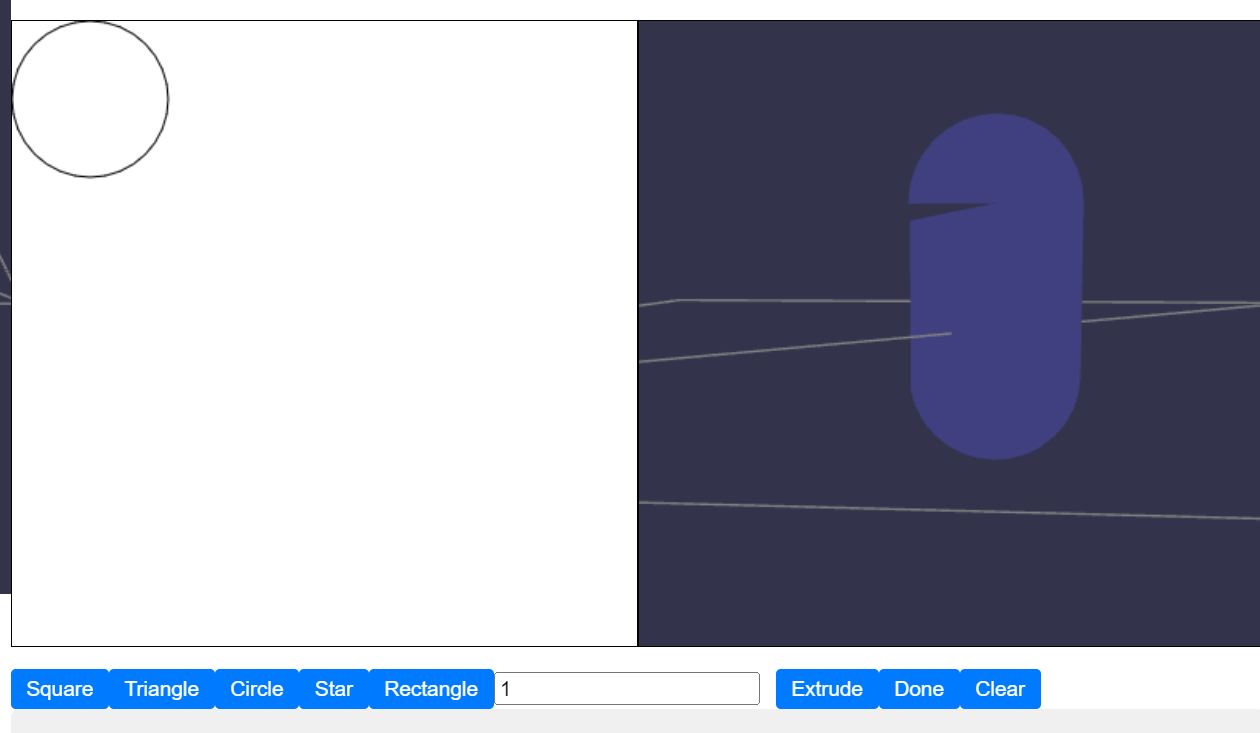
2. Frontend Thought Process:

1. 3D Rendering Performance - Our initial implementation rendered all shapes at full detail, causing performance issues as the number of shapes increased. I had implemented a level-of-detail (LOD) system, reducing the complexity of shapes based on their distance from the camera. This significantly improved frame rates, especially on lower-end devices
2. 2D to 3D Conversion Accuracy- Converting 2D drawn shapes into 3D objects initially resulted in distorted or inaccurate representations. I tried to develop a custom algorithm that uses the 2D points to create a properly scaled and oriented 3D shape, ensuring that the extruded object closely matches the user's 2D drawing.
3. UI Responsiveness and State Management - As the application grew more complex, managing state across components became challenging, leading to inconsistencies and performance issues. I tried to addressed this by implementing a centralized state management system using NgRx, which improved data flow and made the application more predictable and easier to debug.
4. Cross-browser Compatibility – I had encountered significant rendering differences across browsers, particularly with complex shapes. To address this, I implemented a series of feature detection checks and fallback rendering methods

## 5. Results and Screenshots

Achievements:

- Successfully implemented an intuitive 2D shape drawing interface and Achieved smooth and accurate 3D extrusion of drawn shapes.



- Established a foundation for multiplayer interaction.

