A PROJECT REPORT

on

"3D Game Development using Three.js"

Submitted to

KIIT Deemed to be University

In Partial Fulfillment of the Requirement for the Award of

BACHELOR'S DEGREE IN
INFORMATION TECHNOLOGY

BY

KOUSIK CHAKRABORTY **21051902**

UNDER THE GUIDANCE OF
AJAY ANAND

SCHOOL OF COMPUTER ENGINEERING

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

BHUBANESWAR, ODISHA - 751024

April 2023

KIIT Deemed to be University

School of Computer Engineering
Bhubaneswar, ODISHA 751024

CERTIFICATE

This is certify that the project entitled

"DODGE THE BOX"

submitted by

KOUSIK CHAKRABORTY 21051902

is a record of bonafide work carried out by him, in the partial fulfillment of the requirement for the award of Degree of Bachelor of Engineering (Computer Science & Engineering OR Information Technology) at KIIT Deemed to be university, Bhubaneswar. This work is done during year 2022-2023, under our guidance.

Date:13 /4 /2023

(Ajay Anand)
Project Guide

Acknowledgments

I am profoundly grateful to **Ajay Anand** of **Affiliation** for his expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion. .....

Kousik Chakraborty

ABSTRACT

Computer graphics is a rapidly evolving field that has revolutionized the way we interact with and visualize data. This research paper explores the use of computer graphics and Three.js, a popular JavaScript library, in creating immersive graphical applications. The paper discusses the role of computer graphics in various fields such as medicine, virtual reality, and web applications. This research paper delves into the integration of computer graphics using medical data and its implications on education.Additionally, the paper investigates the challenges and advancements in migrating computer graphics to the web, considering standards and technology advances. In today's rapidly changing world, the significance of accurate weather forecasts cannot be overstated.Computer graphics, with its advancements in technologies such as artificial intelligence and smart graphics, has enabled the development of applications that adapt to user requirements and host-machine capabilities.These dynamic capabilities extend the utility of computer graphics across a variety of industries and open up new possibilities for customization and user interaction.

Keywords: **Computer graphics, Three.js, immersive graphical applications, JavaScript library, medicine, virtual reality, web applications.**

Contents

NAME OF PROJECT

Chapter 1

Introduction

Game development in the entertainment industry has a rich history that has evolved significantly over the years. Here is an overview of the history and current trends in game development:

History of Game Development:

Early Years (1950s-1970s): The early years of game development were marked by simple text-based games and early graphical games like "Space war!" developed in the 1960s. These games laid the foundation for the industry.

Arcade Era (1970s-1980s): The rise of arcade games in the 1970s and 1980s, with titles like "Pong," "Space Invaders," and "PAC-Man," brought gaming to a wider audience and established the commercial viability of the industry.

Home Console Revolution (1980s-1990s): The introduction of home consoles like the Atari 2600, Nintendo Entertainment System (NES), and Sega Genesis in the 1980s and 1990s led to a surge in game development for home platforms.

PC Gaming and Rise of 3D Graphics (1990s-2000s): The 1990s saw the rise of PC gaming with titles like "Doom" and "Quake," pushing the boundaries of 3D graphics and gameplay. This era also saw the emergence of game development tools and engines.
Mobile and Indie Game Revolution (2000s-Present): The 2000s marked the rise of mobile gaming with the launch of smartphones and app stores. Indie game development gained popularity, leading to innovative and diverse game experiences.

Current Trends in Game Development:

Virtual Reality (VR) and Augmented Reality (AR): VR and AR technologies have gained traction in game development, offering immersive and interactive experiences. Games like "Beat Saber" and "Pokémon GO" showcase the potential of these technologies.

Live Services and Games as a Service (GaaS): Many game developers are adopting a live service model, providing ongoing updates, events, and content to keep players engaged. Games like "Fortnite" and "Apex Legends" exemplify this trend.

Cross-Platform Play and Cloud Gaming: Cross-platform play allows gamers to play together across different devices, while cloud gaming services like Google Stadia and Microsoft xCloud offer streaming access to games on various platforms.

Esports and Competitive Gaming: Esports has become a significant part of the gaming industry, with professional tournaments, leagues, and a growing audience. Games like "League of Legends" and "Overwatch" are popular in the esports scene.

Accessibility and Inclusivity: Game developers are focusing on making games more accessible and inclusive for all players, including options for customization controls, subtitles, and diverse representation in games.
Artificial Intelligence (AI) and Procedural Generation: AI technologies are being used in game development for various purposes, such as NPC behavior, procedural generation of content, and adaptive difficulty levels.

These trends reflect the dynamic nature of game development in the entertainment industry, driven by technological advancements, changing player preferences, and evolving business models. Game developers continue to push boundaries and innovate to create engaging and immersive gaming experiences for audiences worldwide.Three.js plays a significant role in game development by providing developers with a powerful toolset for creating interactive 3D graphics and games on the web. Here are some key points highlighting the importance of Three.js in game development and the benefits it offers to developers:

Ease of Use: Three.js abstracts the complexities of WebGL programming, making it more accessible for developers to work with 3D graphics in web browsers. It provides a high-level API that simplifies tasks such as rendering, lighting, and camera controls, allowing developers to focus on game logic and design.

Cross-Platform Compatibility: Three.js enables developers to create games that can run on various platforms and devices with web browser support. This cross-platform compatibility eliminates the need to develop separate versions of the game for different operating systems, enhancing the reach and accessibility of the game.

Performance Optimization: Three.js leverages WebGL's hardware acceleration capabilities to deliver high-performance 3D graphics rendering in real-time. Developers can take advantage of optimizations such as shader programming, texture compression, and geometry instancing to achieve smooth gameplay experiences even on lower-end devices.

Rich Visual Effects: Three.js offers a wide range of built-in features and plugins for creating visually stunning effects in games, such as realistic lighting, shadows, reflections, and post-processing effects. Developers can enhance the visual appeal of their games with these effects to create immersive and engaging experiences for players.

Three.js has a large and active community of developers, artists, and enthusiasts who contribute tutorials, examples, and libraries to support game development. Developers can benefit from this community support by accessing resources, sharing knowledge, and collaborating on projects to accelerate their game development process.
Three.js is scalable and flexible, allowing developers to create games of varying complexity and scale, from simple 3D animations to full-fledged multiplayer experiences. Developers can customize and extend Three.js functionality to meet the specific requirements of their game projects, making it suitable for a wide range of game development needs.

Chapter 2

Game Development

Computer graphics theory is foundational to game development, encompassing a range of principles and techniques used to create visual elements in video games. Here are some key areas of research in computer graphics theory related to game development:

1. **Rendering Techniques**: Rendering is the process of generating images from 3D models and scenes. Research in rendering techniques focuses on algorithms for realistic lighting, shadows, reflections, and other visual effects. Techniques such as ray tracing, rasterization, and global illumination play a crucial role in achieving high-quality graphics in games.

2. **Shading Models**: Shading models determine how light interacts with surfaces in a virtual environment. Research in shading models explores methods for simulating materials with different properties, such as diffuse, specular, and glossy reflections. Physically based shading is a popular approach that aims to accurately replicate real-world light behaviour in virtual scenes.

3. **Texture Mapping**: Texture mapping is the process of applying 2D images (textures) to 3D surfaces to enhance their appearance. Research in texture mapping includes advancements in texture synthesis, texture filtering, and texture compression techniques. Procedural texture generation is also an area of interest, allowing developers to create textures algorithmically rather than relying on pre-authored images.

4. **Animation and Rigging**: Animation is essential for bringing characters and objects to life in games. Research in animation focuses on techniques for character rigging, skeletal animation, blend shapes, and motion capture. Real-time animation systems are of particular interest in game development, enabling dynamic and responsive character animations during gameplay.

5. **Level of Detail (LOD) Techniques**: LOD techniques are used to optimize performance by dynamically adjusting the level of detail in 3D models based on factors such as distance from the camera. Research in LOD techniques explores algorithms for efficient model simplification, mesh decimation, and LOD transition management to ensure smooth transitions between levels of detail.

6. **Physics Simulation**: Physics simulation adds realism to game environments by simulating the behaviour of physical objects, such as rigid bodies, soft bodies, fluids, and particles. Research in physics simulation focuses on algorithms for collision detection, rigid body dynamics, fluid dynamics, and other physical phenomena. Integration with rendering systems is also an area of interest to achieve visually coherent simulations.

7. **Real-Time Graphics Optimization**: Real-time graphics optimization techniques are essential for maintaining smooth and responsive gameplay experiences on various hardware platforms. Research in optimization includes algorithms for efficient rendering, culling techniques, occlusion culling, and resource management strategies to maximize performance while maintaining visual quality.

8. **Emerging Technologies**: Emerging technologies such as virtual reality (VR), augmented reality (AR), and real-time ray tracing are driving innovation in game graphics. Research in these areas explores new rendering techniques, interaction paradigms, and hardware advancements to support immersive and visually stunning gaming experiences.

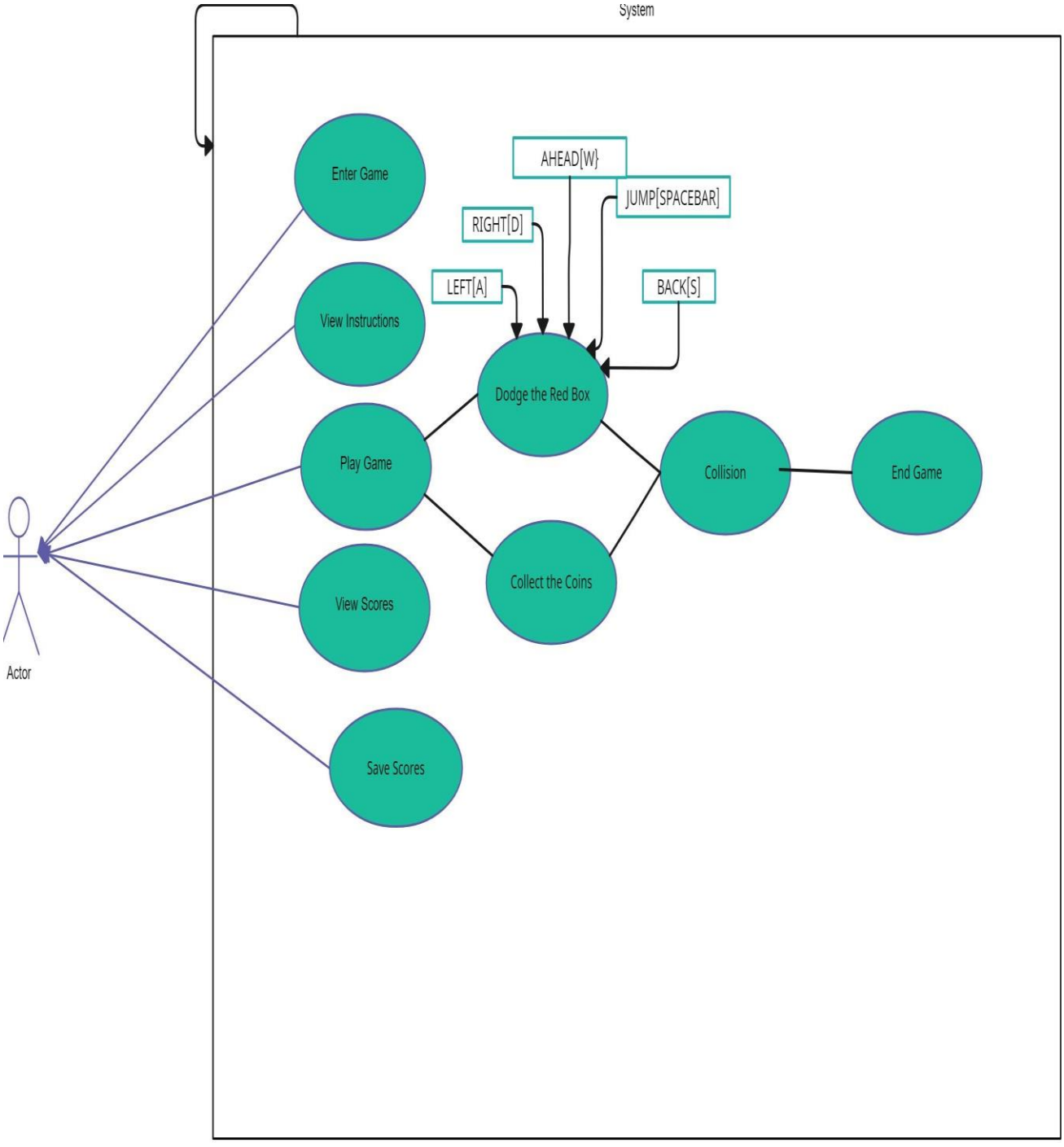
By delving into these areas of research, game developers can gain a deeper understanding of computer graphics theory and leverage advanced techniques to create visually compelling and engaging games.

Chapter 3

Requirements & Specifications (SRS)

"**Dodge the Box**" is an immersive 3D game developed using Three.js, where players control a box moving along a path filled with obstacles. The goal is to navigate the box safely through the treacherous path, avoiding collisions with obstacles and reaching the end of the level.

3.1 Use Case



3.2 Game Description

3.2.1 Product Perspective

"Dodge the Box" caters to a broad audience, from casual gamers to anyone seeking a quick and challenging 3D experience. No prior knowledge of Three.js is necessary.

3.2.2 Product Functions

Immersive 3D World: Three.js creates a visually stunning 3D environment for your gameplay.
Box Control: You command a box, guiding it along a predefined path using the arrow keys (up, down, left, right).
Obstacle Challenge: Strategically placed obstacles (cubes, spheres) test your reflexes and require precise maneuvering.
Collision Detection: Collisions with obstacles result in a game over or point deduction (depending on difficulty).
(Optional) Scoring System: Track your success by earning points for skillfully navigating the path without collisions.
(Optional) User Interface: A user interface (UI) can be implemented to display the score, difficulty level, and other relevant information.

3.2.3 User Classes and Characteristics

The game is designed for a general audience with basic computer literacy and familiarity with keyboard controls.

3.2.4 General Constraints

The game will run in a web browser using Three.js.
Performance optimization will ensure smooth gameplay on various devices.

3.2.5 Assumptions and Dependencies

Users have a web browser with WebGL support.
Basic understanding of keyboard controls is assumed.

3.3 Specific Requirements

3.3.1 Functional Requirements

3.3.1.1 Core Gameplay Mechanics:

- **Scene Creation:** Three.js establishes a 3D scene with a captivating background color. A perspective camera allows for a dynamic view of the game world.

- **Box Creation:** A cube geometry represents the player-controlled box. A chosen material defines the box's visual appearance (color, shading). The geometry and material are combined into a mesh object, then added to the scene.

- **Path Definition**: The path is defined as a series of points in 3D space stored in an array of THREE.Vector3 objects. The box's position updates along these points based on movement speed and user input.

- **Obstacle Creation:** Obstacle geometries (cubes, spheres) are created using Three.js primitives. Obstacles are positioned at various points along the path for a challenging experience. Each obstacle is converted into a mesh and added to the scene.

- **Collision Detection**:Three.js's built-in bounding spheres (THREE.BoundingSphere) are used for collision detection.The game continuously checks for collisions between the box and obstacles.

- **Input Handling:** Event listeners capture keyboard arrow presses (up, down, left, right) for controlling the box's movement.

- **Game Loop:** A continuous animation loop is established using requestAnimationFrame. Within the loop: Update the box's position based on user input and path definition. Perform collision detection between the box and obstacles.Handle collisions (e.g., stop movement, display game over message, or deduct points). Render the scene using renderer.render(scene, camera).

3.3.1.2 Enhancements :

Scoring System:
Implement a mechanism to track points awarded for successfully navigating the path without collisions.
Display the score on the screen, allowing players to track their progress and compete with others (future feature).
Difficulty Levels:
Cater to various skill levels by offering different difficulty options. This could involve:
Increased number or density of obstacles.
Faster box movement speed.Time limits for completing the path.

User Interface:

- Design a user interface to display:
- Current score.
- Selected difficulty level.
- Timer (if applicable).
- Game over or victory message.
- Ensure the UI is clear, concise, and visually appealing

3.3 Non-functional requirement:

Performance:

The application render 3D scenes efficiently, utilizing techniques such as occlusion culling, level-of-detail (LOD) rendering, and geometry instancing to optimize rendering performance.
Load times for 3D models and textures is minimized through compression and streaming techniques, ensuring a smooth user experience.
Frame rates consistent and high, aiming for a target frame rate suitable for the intended platform and device.

Scalability:

The system architecture designed to scale horizontally and vertically to accommodate increasing user demand and larger datasets.
Load balancing mechanisms is implemented to distribute incoming traffic across multiple servers or instances to prevent overloading.
Database and storage solutions is scalable to handle growing data volumes efficiently.

Compatibility:

The application is tested across a wide range of devices (desktops, laptops, tablets, smartphones) and browsers (Chrome, Firefox, Safari, Edge, etc.), ensuring compatibility and consistent performance.
Compatibility with various operating systems (Windows, macOS, Linux, iOS, Android) should also be ensured.

Security:

User authentication and authorization mechanisms is implemented to control access to sensitive features and data within the application.
Encryption techniques such as HTTPS/TLS is used to secure data transmission between the client and server.
Input validation and sanitization is enforced to mitigate common security threats such as SQL injection and cross-site scripting (XSS).
Regular security audits and penetration testing is conducted to identify and address vulnerabilities.

Reliability:

The system is highly available and resilient to failures, with built-in redundancy and failover mechanisms to minimize downtime.
Error monitoring and logging is implemented to track and diagnose issues, allowing for timely resolution of problems.
Automated testing, including unit tests, integration tests, and end-to-end tests, is conducted to validate system behavior and prevent regressions.

Availability:

The system is designed with high availability in mind, with redundant components and fail over systems to ensure continuous operation.
Scheduled maintenance activities is performed during off-peak hours to minimize disruption to users.
Service Level Agreements (SLAs) is established to define acceptable levels of uptime and response times.

Usability:

The user interface is designed with usability principles in mind, featuring intuitive navigation, clear labeling, and consistent design patterns.
User feedback mechanisms such as tooltips, error messages, and progress indicators should be provided to guide users and enhance usability.
Accessibility features such as keyboard navigation, screen reader support, and adjustable font sizes is implemented to accommodate users with disabilities.

Maintainability:

The codebase is adhere to coding standards and best practices, with clear and concise code that is easy to understand and maintain.
Documentation is comprehensive and up-to-date, covering code structure, APIs, and development guidelines.
Version control systems (e.g., Git) is used to manage changes to the codebase, facilitating collaboration and versioning.

Performance Efficiency:

Resource usage is optimized to minimize memory consumption, CPU utilization, and network bandwidth usage.
Caching mechanisms should be employed to reduce the need for redundant computations and data retrieval.
Resource cleanup and garbage collection is performed efficiently to reclaim unused memory and resources.

Data Integrity:

Data validation and integrity checks is implemented at various stages of data processing to prevent data corruption or loss.
Backup and recovery mechanisms is in place to protect against data loss due to hardware failures, human error, or malicious activities.
Data encryption techniques is used to protect sensitive data at rest and in transit.

Chapter 4

Tools & Technologies Used

4.1 Core Technologies

- **HTML (HyperText Markup Language)**: HTML provides the fundamental structure of the game's web interface. While its role in this example is minimal, HTML is crucial in larger projects for defining elements like buttons, score displays, menus, and more.

- **CSS (Cascading Style Sheets)**: CSS is used to style the visual presentation of the game. The example code includes basic background and element styling, but CSS can extensively control the look and feel, from colors and fonts to layout and animation effects.

- **JavaScript** : JavaScript is the programming language driving the game's logic, behaviors, and interactions. It handles the following:

- **Object Creation and Manipulation:** Creating game objects (cubes, ground, enemies) and defining their properties (size, color, position, velocity).
- **Game Logic:** Implementing core game mechanics, such as movement, collision detection, gravity, scoring, and win/loss conditions.
- **Event Handling:** Responding to user input (keyboard presses) to control the game character.
- **Animation and Updates:** Driving the game's dynamic nature using JavaScript's `requestAnimationFrame()` function for smooth rendering.

- **Three.js** : Three.js is a powerful JavaScript library specializing in creating and rendering 3D graphics within a web browser. It is the heart of the game's visual presentation.

Key roles of Three.js in this project:

- **Scene Setup:** Creating a THREE.Scene object, acting as the container for all 3D objects.
- **Camera:** Establishing a THREE.PerspectiveCamera to define the player's viewpoint in the 3D world.
- **Objects (Geometries and Materials):** Building 3D shapes (THREE.BoxGeometry) and applying textures or colors (THREE.MeshStandardMaterial) to create the cube, ground, and enemies.
- **Lighting:** Adding THREE.DirectionalLight and THREE.AmbientLight to illuminate the scene, making objects appear three-dimensional and enhancing visual depth.
- **Rendering:** Utilizing the THREE.WebGLRenderer to render the constructed 3D scene onto the web page.
- **Physics (Basic):** Enabling basic physics-like behaviors with gravity and collision detection.
- **OrbitControls (Extension):** The OrbitControls extension from Three.js allows the player to rotate and manipulate the camera view for better exploration of the 3D environment.

Additional Development Tools

- **Text Editor / IDE:** A code editor (like Visual Studio Code, Sublime Text, or Atom) or a full-fledged Integrated Development Environment (IDE) is necessary for writing, debugging, and organizing the HTML, CSS, and JavaScript code.
- **Web Browser** : A modern web browser (Chrome, Firefox, Edge, etc.) is used to run and test the game during development as well as by the end user.
- **Browser Developer Tools:** The web browser's built-in developer tools are invaluable for debugging, inspecting elements, profiling performance, and analyzing network requests.

4.2 Enhancements and Further Exploration

Advanced Physics Engine

Why It Matters: Basic physics in the current code are functional but quite limited. A dedicated physics engine opens up significantly greater possibilities for a richer gameplay experience.

Benefits of Cannon.js or Ammo.js:

- Realistic Interactions: Simulate forces like friction, drag, and realistic bouncing for objects, greatly improving how the game feels.
- Complex Collisions: Accurately detect collisions between complex shapes (not just boxes), enabling more intricate level design and object interactions.
- Constraints: Implement joints, hinges, and other constraints to create connected objects like chains, vehicles with wheels, or ragdoll effects.
- Integration:

- These libraries often work by creating invisible physics bodies associated with your visible 3D meshes.
- Each frame, you'd update the position and rotation of your Three.js objects based on the results of the physics simulation.

Textures and Models

Why It Matters: Currently, the game uses simple geometric shapes. More detailed models and textures dramatically enhance visual appeal and immersion.

3D Model Formats:

- .obj: A classic format widely supported by modeling software. Often accompanied by .mtl (material) files.
- .gltf: A modern, efficient format optimized for web delivery. Can include embedded textures, animations, and scene hierarchy information.
- Texturing: Applying image-based textures to your models will add surface detail such as bricks, wood grain, metal, or even character skin.

- Three.js has loaders for common model formats and tools for applying textures to materials.

Sources of 3D Models:

- Create your own: Use modeling software like Blender (free).
- Online Repositories: Sites like Sketchfab, TurboSquid, or the Unity Asset Store offer a wealth of free and paid models.

Sound Effects and Music

Why It Matters: Audio elevates the experience, providing crucial feedback, setting atmosphere, and enhancing emotional engagement.

Web Audio API:

- Allows loading and playing sound files.
- Enables dynamic effects (e.g., changing pitch based on object speed, 3D spatial audio).
- Audio Libraries: Libraries like Howler.js can simplify sound management and playback controls.

Types of Audio:

- Sound Effects: Actions like jumping, collisions, item collection.
- Ambient Background: Nature sounds, wind, machinery, etc., to enhance scene setting.
- Music: Background tracks to establish a mood or increase tension.

4.3 Game Design and Mechanics

Why It Matters: Engaging gameplay is what keeps players coming back. Building upon the core is key.

Enemy Behaviors:

- Instead of just moving forward, enemies could pattern, chase the player, or exhibit unique attack behaviors.
- Consider using path finding techniques to make enemies navigate the level intelligently.

Power-ups:

- Temporary speed boosts, invincibility, special abilities create exciting strategic moments.
- Level Progression: Introduce multiple levels with increasing difficulty and variations in scenery or obstacles.
- Design challenges that demand the player master different game mechanics.

Scoring System:

- Reward players based on time, enemies defeated, items collected, etc.
- Incorporate high scores or leader boards to add a competitive element.

Important Considerations:

- Performance: Complex models, textures, and physics calculations can impact frame rate. Prioritize optimization techniques.
- Asset Sources: Be aware of licensing when using external models, sounds, or music.
- Game Design is Key: Technology is a tool; focus on making your game fun, challenging, and rewarding.

Chapter 5

System Design

CLASSES:

Box

Property	***Description***
width	**Width of the box**
height	**Height of the box**
depth	**Depth of the box**
color	**Color of the box**
velocity	**Object representing velocity in x, y, and z directions**
position	**Object representing position in x, y, and z coordinates**
zAcceleration	**Boolean indicating if z acceleration is enabled**

Method	***Description***
constructor	**Initializes the Box object with specified parameters**
updateSides	**Updates the side me**
update	**Updates the position of the box based on velocity and collisions**
applyGravity	**Applies gravity to the box and handles collision with ground | asurements of the box**

Coin

Property	***Description***
radius	**Radius of the coin**
color	**Color of the coin**
position	**Object representing position in x, y, and z coordinates**
rotationSpeed	**Speed of rotation for the coin**

Method	***Description***
constructor	**Initializes the Coin object with specified parameters**
updateRotation	**Updates the rotation of the coin**

Fig1. THREE.js Geometry

THREE.Scene:
This represents a scene, which is like a container that holds all objects, cameras, and lights. It's where all the action happens in your 3D world.

THREE.PerspectiveCamera:
This is a type of camera that mimics the behavior of a real-world camera. It's used to define what is visible in the scene from a specific viewpoint and perspective.

THREE.Mesh:

This is a type of object in Three.js that represents a geometric shape (like a cube or sphere) wrapped in a material. Meshes are the building blocks of most 3D scenes.

THREE.BoxGeometry, THREE.SphereGeometry:
These are predefined geometries that define the shape of objects. For example, BoxGeometry creates a cube, and SphereGeometry creates a sphere. They are used to define the shape of your Box and Coin objects.

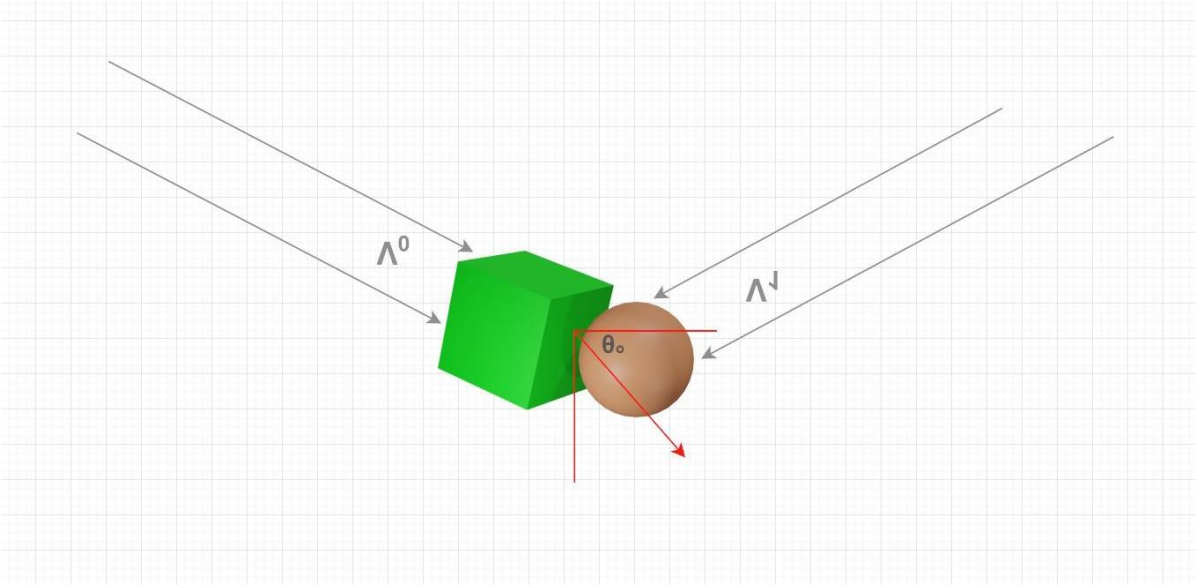
THREE.MeshStandardMaterial:
This is a type of material that defines the appearance of a mesh. It includes properties like color, roughness, and metalness. In your code, it's used to give color to the Box and Coin objects.

THREE.DirectionalLight, THREE.AmbientLight:
These are types of lights in Three.js. DirectionalLight simulates light that comes from a specific direction, like sunlight. AmbientLight provides overall, non-directional lighting to the scene, simulating the light that fills an entire room.

THREE.DirectionalLightHelper:
This helper object is used for visualizing directional lights in the scene. It helps to see the direction and intensity of the light.

THREE.Audio:
This represents an audio source in Three.js. It's used to load and play sounds in your scene.

Fig2. Collision Detection



Chapter 6
Implementation

IMPORTED THREEJS PACKAGE

```
<style>
   body {
     margin: 0;
     background: #0c4a6e ;
   }
 </style>

 <script
  async
  src="https://unpkg.com/es-module-shims@1.6.3/dist/es-module-shims.js"
 ></script>

 <script type="importmap">
  {
    "imports": {
      "three": "https://unpkg.com/three@0.150.1/build/three.module.js",
      "three/addons/": "https://unpkg.com/three@0.150.1/examples/jsm/"
    }
  }
 </script>
```

SET CAMERA POSITION

```
<script type="module">
   import * as THREE from 'three'
   import { OrbitControls } from 'three/addons/controls/OrbitControls.js'

   const scene = new THREE.Scene()
   const camera = new THREE.PerspectiveCamera(
     75,
     window.innerWidth / window.innerHeight,
     0.1,
     1000
   )
   camera.position.set(
      3.9, 3.5, 8.5);
```

DEFINED CLASS AND INITIALIZED VIA CONSTRUCTOR

```
class Box extends THREE.Mesh{
    constructor({
        width,
        height,
        depth,
        color = '#00ff00',
        velocity={
            x:0,
            y:0,
            z:0
        },
        position={
            x:0,
            y:0,
            z:0
        },
        zAcceleration = false    })
```

DECLARE POSITIONING VARIABLES

```
{
        super(new THREE.BoxGeometry(width,height,depth),new
THREE.MeshStandardMaterial({ color: color }))
        this.width=width
        this.height=height
        this.depth=depth

        this.position.set(position.x, position.y, position.z)  //can't directly set "this.position" since it
is a read only property of THREE.Mesh hence "this.position.set()"

        this.right = this.position.x + this.width/2
        this.left = this.position.x - this.width/2

        this.bottom= this.position.y - this.height / 2
        this.top = this.position.y + this.height/2

        this.front = this.position.z - this.depth/2
        this.back = this.position.z + this.depth/2

        this.velocity = velocity
        this.gravity = -0.004

        this.zAcceleration = zAcceleration

    }
```

DEFINED SIDE MEASUREMENTS OF THE BOX

```
updateSides(){
        this.bottom= this.position.y - this.height / 2
        this.top = this.position.y + this.height / 2

        this.right = this.position.x + this.width/2
        this.left = this.position.x - this.width/2

        this.front = this.position.z - this.depth/2
        this.back = this.position.z + this.depth/2
    }
```

DEFINED GROUND MEASUREMENTS

```
update(ground){
    this.updateSides()

    if(this.zAcceleration){this.velocity.z  +=0.001}

    this.position.x += this.velocity.x
    this.position.z += this.velocity.z
    // if(xCollision){
    //     console.log("collision detected left")
    // }
    // if(this.left <= ground.right){
    //     console.log("collision detected right")
    // }
    // if(this.front <= ground.back){
    //     console.log("collision detected back")
    // }
    // if(this.back >= ground.front){
    //     console.log("collision detected front")
    // }

    this.applyGravity(ground)
}
```

APPLIED GRAVITY ON THE BOX

```
applyGravity(ground){
    this.velocity.y += this.gravity
    // this.position.y += this.velocity.y

    if(boxCollision({
        box1:this,
        box2:ground}))
    {
        const friction = 0.5
        this.velocity.y *= friction
        this.velocity.y= -this.velocity.y}
    else this.position.y += this.velocity.y
    }
}
```

CODE TO CHECK FOR BOX COLLISION

```
function boxCollision({box1,box2}){//checking for boundary overlap
    //detect for collision
    const xCollision = (box1.right >= box2.left) && (box1.left <= box2.right);
    const zCollision = (box1.front <= box2.back) && (box1.back >= box2.front);
    const yCollision = (box1.bottom + box1.velocity.y <= box2.top) && (box1.top >=
box2.bottom);

    return (xCollision && yCollision && zCollision)
}
```

DEFINED CLASS AND CONTRUCTOR FOR COIN

```
class Coin extends THREE.Mesh {
        constructor({
            radius = 0.5,
            color = '#ffff00',
            position = {
                x: 0,
                y: 0,
                z: 0
            }


        })
```

APPLIED ROTATION TO COIN

```
{
            super(new THREE.SphereGeometry(radius, 32, 32), new
THREE.MeshStandardMaterial({ color: color }));
            this.radius = radius;
            this.position.set(position.x, position.y, position.z);




// Apply rotation animation
            this.rotationSpeed = Math.random() * 0.1; // Random rotation speed
        }


updateRotation() {
            this.rotation.y += this.rotationSpeed;
        }
    }
```

CODE FOR SPAWNING COINS

```
const coins = [];

    function spawnCoin() {
        const coin = new Coin({
            radius: 0.5,
            color: '#ffff00',
            position: {
                x: (Math.random() - 0.5) * 10, // Random x position within gameplay area
                y: 0, // Adjust this value based on the height of your player box
                z: -Math.random() * 20 // Adjust this value based on how far ahead the player can see
            }
        });
        scene.add(coin);
        coins.push(coin);
    }
```

v

ADDED SOUND TO COIN COLLECTION

```
function playCoinSound() {
        const audio = new Audio('coin_sound.mp3'); // Replace 'coin_sound.mp3' with your coin sound file
        audio.play();

// Play coin collection sound
        playCoinSound();
    }
```

CHECK COIN COLLISION AND KEEP COUNT AND UPDATE SCORE AS PER COIN COLLISION

```
let score = 0;
    function checkCoinCollision() {
        coins.forEach((coin, index) => {
            if (boxCollision({
                box1: cube,
                box2: coin
            })) {
                scene.remove(coin);
                coins.splice(index, 1);
                score++;
                updateScoreDisplay();
            }
        });
    }

    function updateScoreDisplay() {
        document.getElementById("score-display").innerText = "Score: " + score;
    }
```

DEFINED CUBE AND CASTED SHADOW

```
  const cube = new Box({ //creating a new class so that we can have parameters like height depth and width in a cleaner way
    width: 1,
    height: 1,
    depth: 1,
    velocity:{
        x:0,
        y:-0.01,
        z:0
    }
  })
  cube.castShadow=true
  scene.add(cube)
```

DEFINED GROUND AND RECEIVED SHADOW TO PLANE

```
const ground = new Box({
    width:10,
    height:0.5,
    depth:50,
    color:'#0369a1',
    position:{
        x:0,
        y:-2,
        z:0
    }
})


ground.receiveShadow=true
scene.add(ground)
```

ADDED LIGHT FOR SHADOW

```
const light = new THREE.DirectionalLight(0xffffff,1)
    light.castShadow=true
    light.position.z=1
    light.position.y=3
    scene.add(light)

    scene.add(new THREE.AmbientLight(0xffffff, 0.5))

    camera.position.z = 5
```

ADDED CONTROL KEYS FOR THE MAIN BOX USING ‘W A S D’ KEYS AND ADDED EVENT LISTENER

```
const keys={
    w:{pressed:false},
    a:{pressed:false},
    s:{pressed:false},
    d:{pressed:false}
}

window.addEventListener('keydown',(event) =>{
    switch(event.code){
        case 'KeyW':
            keys.w.pressed=true
            break;
        case 'KeyA':
            keys.a.pressed=true
            break;
        case 'KeyS':
            keys.s.pressed=true
            break;
        case 'KeyD':
            keys.d.pressed=true
            break;
        case 'Space':
            cube.velocity.y=0.12;
            break;
    }
})

window.addEventListener('keyup',(event) =>{
    switch(event.code){
        case 'KeyW':
            keys.w.pressed=false
            break;
        case 'KeyA':
            keys.a.pressed=false
            break;
        case 'KeyS':
            keys.s.pressed=false
            break;
        case 'KeyD':
            keys.d.pressed=false
            break;
    }
})
```

SPWANED ENEMIES AND COINS

```
const enemies = [];

  let jumpallow = true;
  let frames = 0;
  let framerate = 100;
  let coinSpawnRate = 200; // Initial spawn rate
```

ANIMATED THE MOVEMENT OF THE ENEMY BOXES

```
  function animate() {
    // console.log(camera.position.x , camera.position.y, camera.position.z);
   frames++;
   if(framerate>10){
    framerate--;
   }
   const animationFrameID = requestAnimationFrame(animate)
   renderer.render(scene, camera)

   enemies.forEach(enemy => {
    enemy.update(ground)
    if (boxCollision({
      box1: cube,
      box2: enemy
    })){
      cancelAnimationFrame(animationFrameID)
    }
   })
```

CODE FOR COIN SPAWN WITH A FIXED RATE

```
if (frames % coinSpawnRate === 0) {
          spawnCoin();
          coinSpawnRate = Math.max(100, coinSpawnRate - 5); // Increase frequency over time
        }

        coins.forEach(coin => {

                                                                                    c
oin.update(ground);
          coin.updateRotation();
        });
```

CODE FOR BOX SPAWN

```
 if(frames % framerate === 0){
      const enemy = new Box({ //creating a new class so that we can have parameters like height depth and width in a cleaner way
      width: 1,
      height: 1,
      depth: 1,
      velocity:{
          x:0,
         y:-0.01,
         z:0.01
      },
      position:{
         x:( Math.random()-0.5 )* 10,
         y: 0,
         z: -20
      },
      color:'red',
      zAcceleration:true
   })
   enemy.castShadow=true
   scene.add(enemy)
      enemies.push(enemy);
   }
```

ADJUSTED VELOCITY AS PER INPUT OF MOVEMENT KEYS

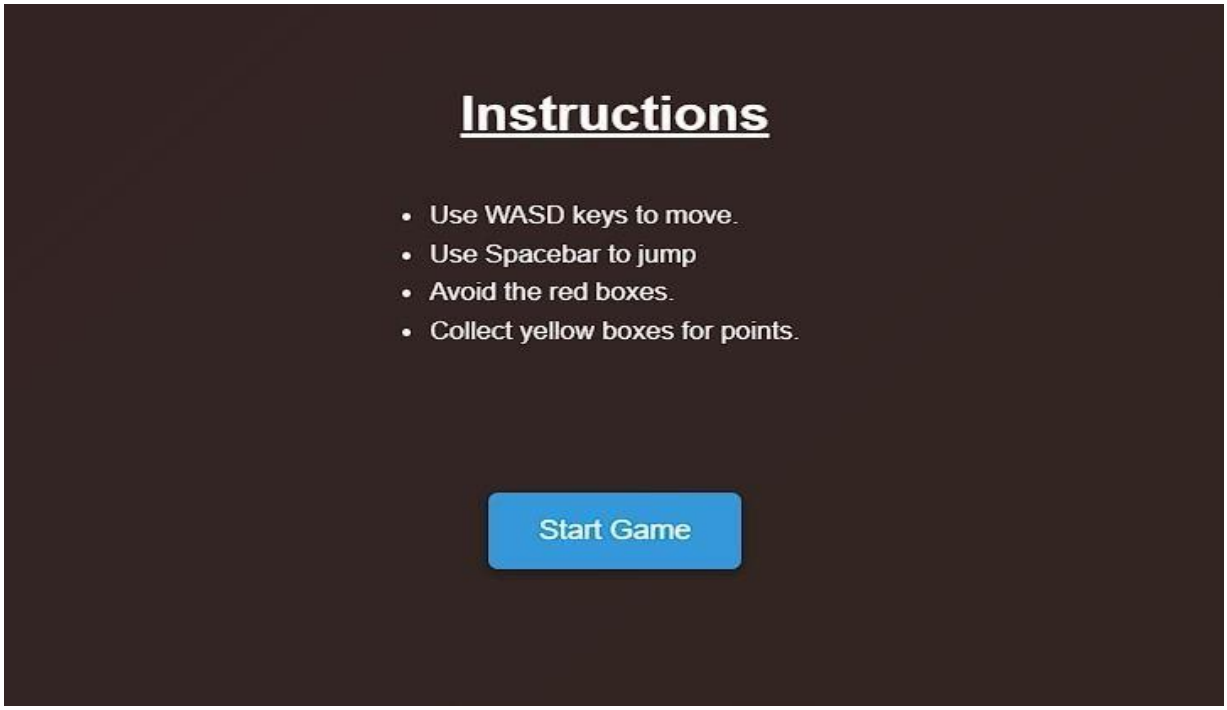
```
  cube.velocity.x=0
    cube.velocity.z=0

   if(keys.a.pressed) cube.velocity.x = -0.02
   else if(keys.d.pressed) cube.velocity.x = 0.02
   if(keys.s.pressed) cube.velocity.z = 0.02
   else if(keys.w.pressed) cube.velocity.z = -0.02
   cube.update(ground)
   //   cube.position.y += -0.01
   //   cube.rotation.x += 0.01
   //   cube.rotation.y += 0.01

   }
   animate()
```

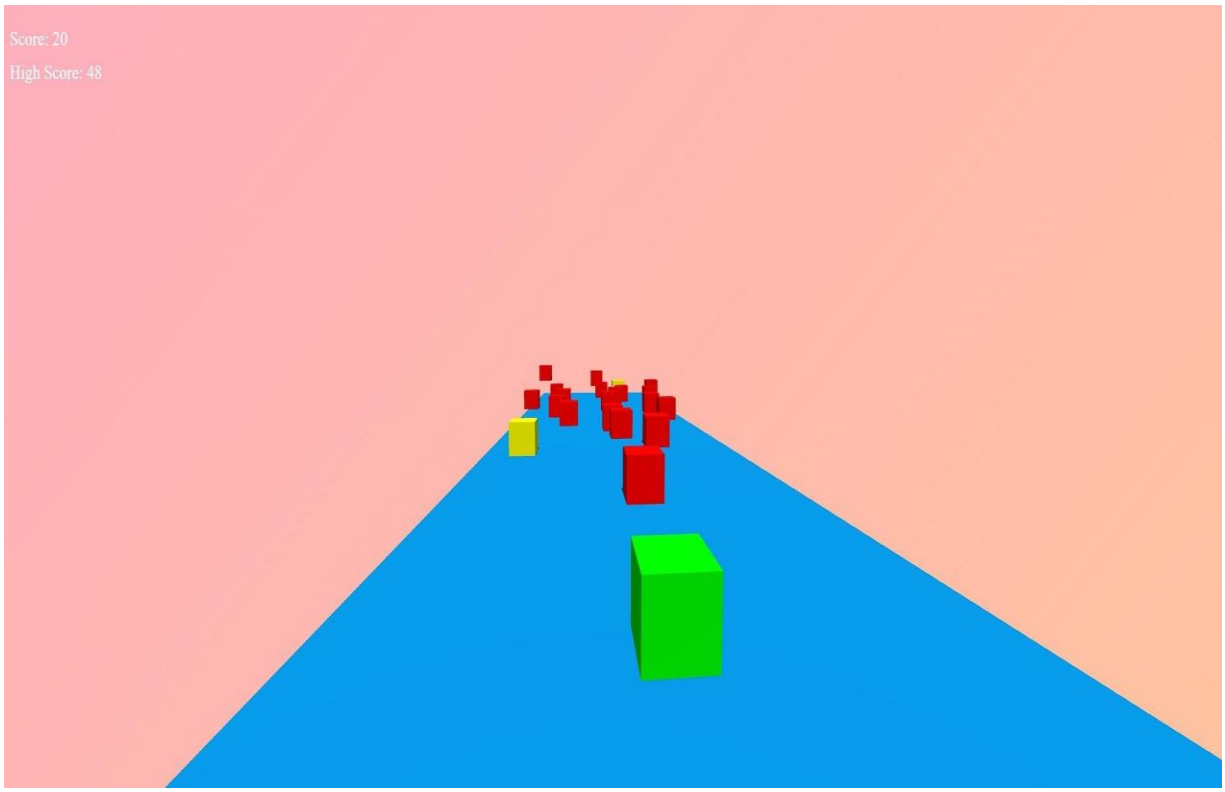
DISPLAYED SCORE

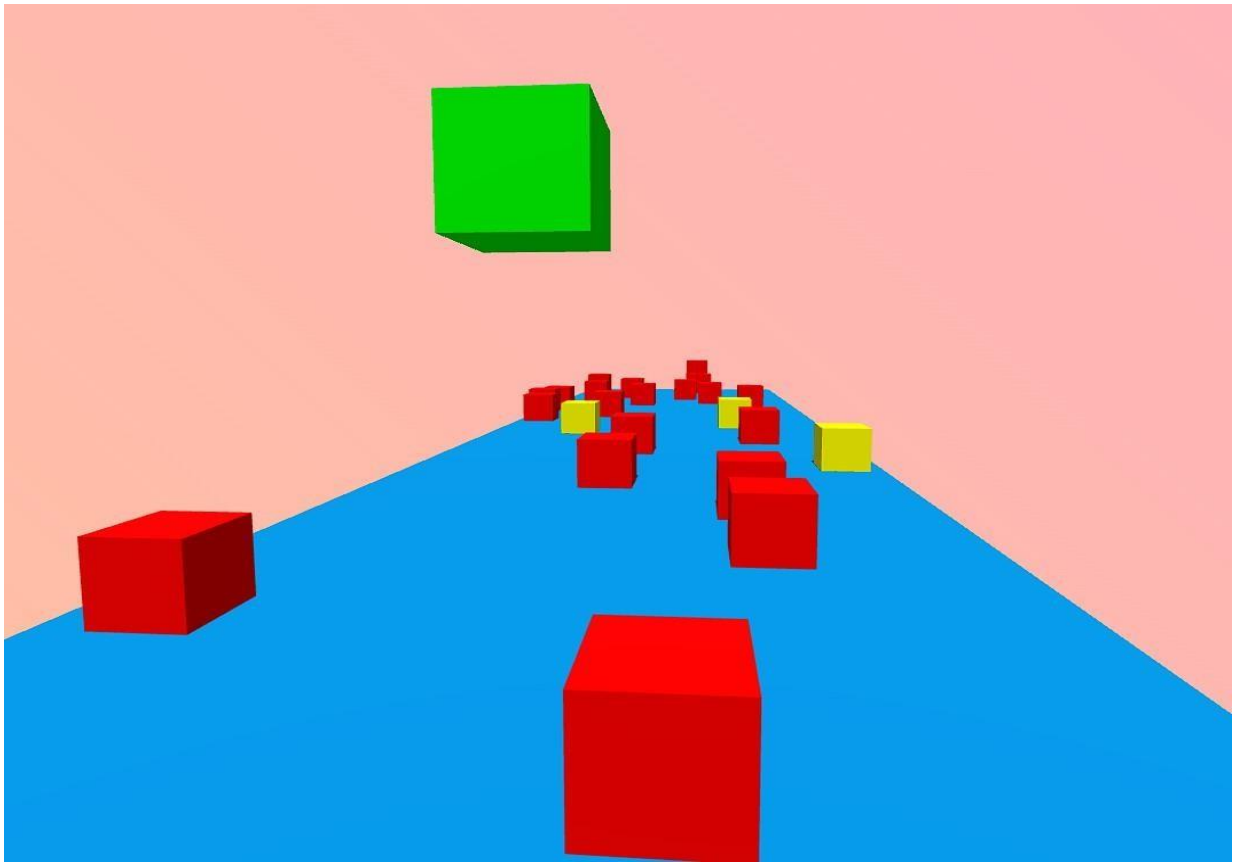
```
<div id="score-display" style="position: absolute; top: 10px; left: 10px; color: white; font-size: 24px;"></div>
```

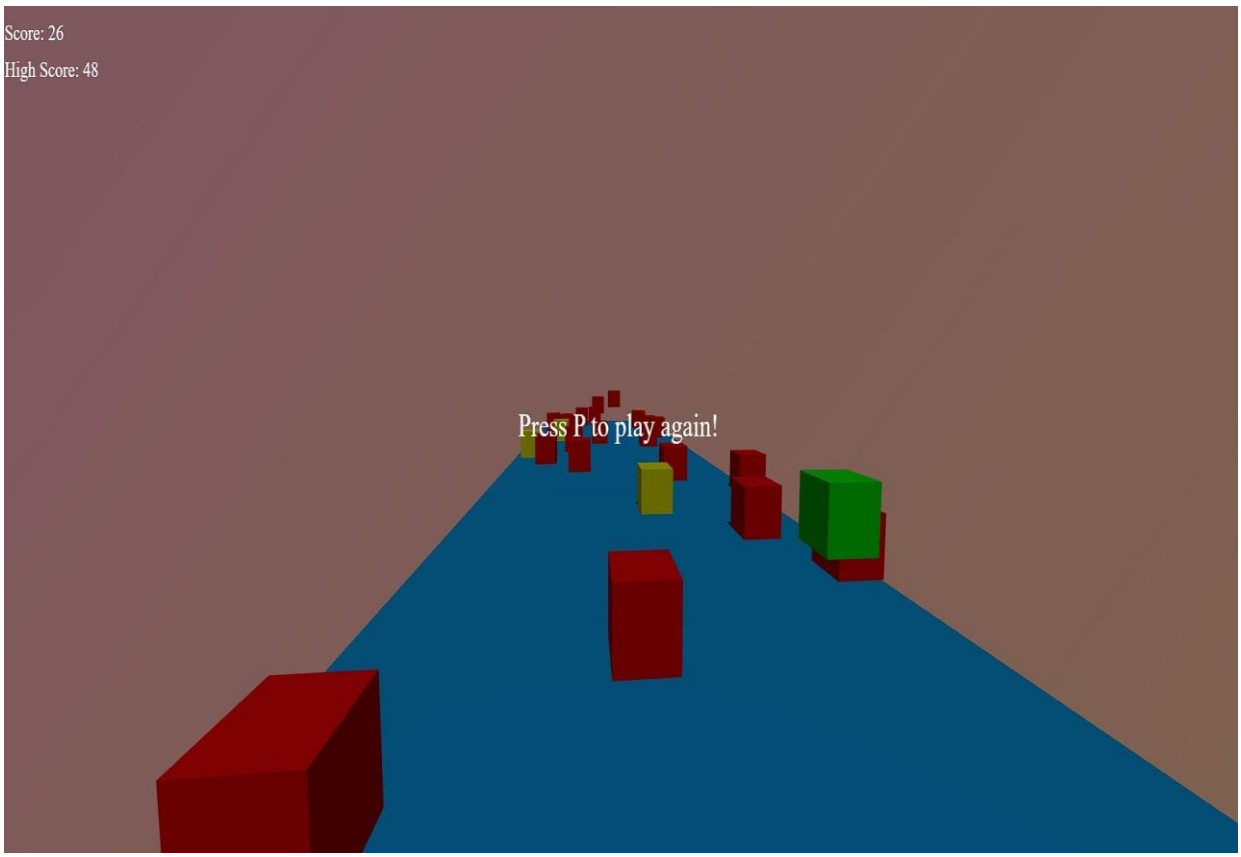
Chapter 7

Project Snapshots








Score: 26
High Score: 48
Press P to play again!

Chapter 8

Conclusion and Future Scope

8.1 Future Scope

1. **Feature Expansion:** Continuously adding new features such as power-ups, obstacles, and challenges to keep the gameplay fresh and engaging. This could involve introducing new environments, such as different planets or fantasy worlds, each with its own unique obstacles and mechanics.

2. **Level Design:** Creating additional levels with varying difficulties and themes to provide players with a diverse and challenging experience. This could include both single-player levels and multiplayer arenas, each designed to test different skills and strategies.

3. **Multiplayer Integration:** Implementing multiplayer modes to allow players to compete against each other in real-time matches. This could include both competitive and cooperative multiplayer modes, such as racing against each other to reach the finish line or working together to overcome obstacles.

4. **Character Customization:** Allowing players to customize their characters with different outfits, accessories, and visual enhancements. This could include unlocking new customization options through gameplay achievements or in-game currency, providing players with a sense of progression and personalization.

5. **Ball Customization:** Similarly, offering options for customizing the ball itself with different colors, patterns, and designs. This could add a layer of personalization to the gameplay experience and allow players to express themselves creatively.

6. **Virtual Reality Integration:** Exploring the integration of virtual reality (VR) technology to provide players with an immersive and interactive gaming experience. This could involve adapting the game mechanics and controls for VR devices, allowing players to physically interact with the game world and feel like they're part of the action.

7. **Partnerships and Collaborations:** Partnering with other brands, franchises, or content creators to introduce crossover content or themed events within the game. This could help expand the game's audience reach and attract fans from different communities.

8. **Community Engagement:** Actively engaging with the player community to gather feedback, suggestions, and ideas for future updates and improvements. This could involve hosting community events, contests, and surveys to foster a sense of community involvement and ownership over the game's development.

8.2 Conclusion

"Dodge the Ball Game" transports players into a vibrant world where they take on the role of a lovable dodge character on an epic adventure. Set against a backdrop of colorful landscapes and dynamic environments, players are immediately drawn into the game's immersive experience.

The premise of "Dodge the Ball Game" is simple yet captivating: guide the dodge through a series of increasingly challenging obstacles while collecting as many balls as possible. The game's intuitive controls make it accessible to players of all ages, allowing them to dive right into the action without any prior experience necessary.

As players progress through the game, they'll encounter a variety of obstacles ranging from simple barriers to intricate mazes and moving platforms. Each level presents a new set of challenges, testing the player's reflexes, timing, and strategic thinking.

One of the game's standout features is its charming visual style, which combines adorable character designs with vibrant colors and lively animations. From the playful expressions of the dodge to the whimsical backgrounds, every aspect of the game is designed to delight and entertain.

In addition to its engaging gameplay and charming aesthetics, "Dodge the Ball Game" offers plenty of replay value. Players can strive to beat their high scores, unlock achievements, and discover hidden secrets scattered throughout the game.

Overall, "Dodge the Ball Game" is more than just a mobile app – it's an immersive gaming experience that captivates players from start to finish. Whether you're a casual gamer looking for some lighthearted fun or a seasoned player seeking a new challenge, "dodge the Ball Game" promises hours of entertainment and enjoyment.

References